Application of the DSNN for the VHbb analysis and tactics to combat computing challenges

Fang-Ying Tsai

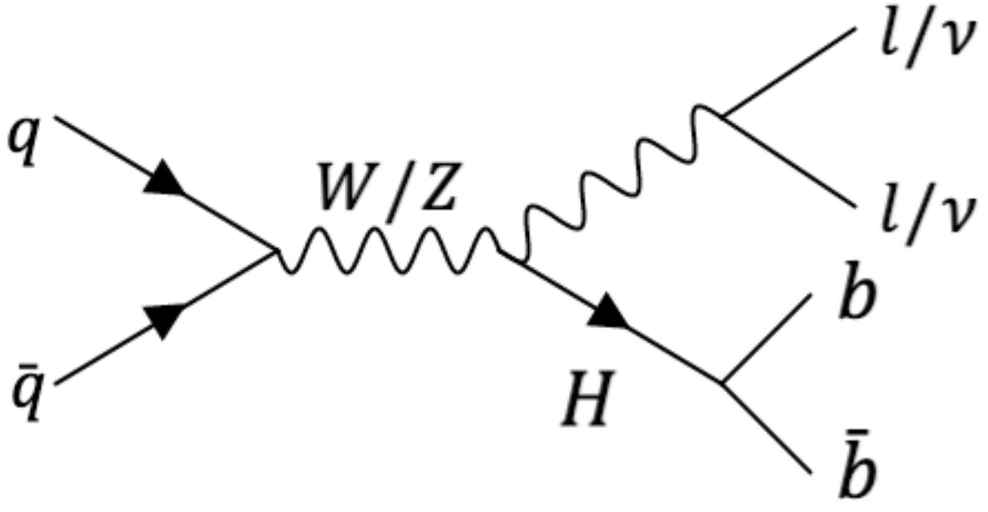
28 January 2022, NPPS meeting

Project Participants for the VH(bb) analysis and Google-ATLAS R&D

John Hobbs, Giacinto Piacquadio,
Torre Wenaus, Alexei Klimentov

Stony Brook University BROOKHAVEN NATIONAL LABORATORY ATLAS EXPERIMENT

Outline



- 3 channels in the VHbb studies:

0 lepton (Z→vv), **1 lepton** (W→lv), 2 leptons(Z→ll) where l = e, µ

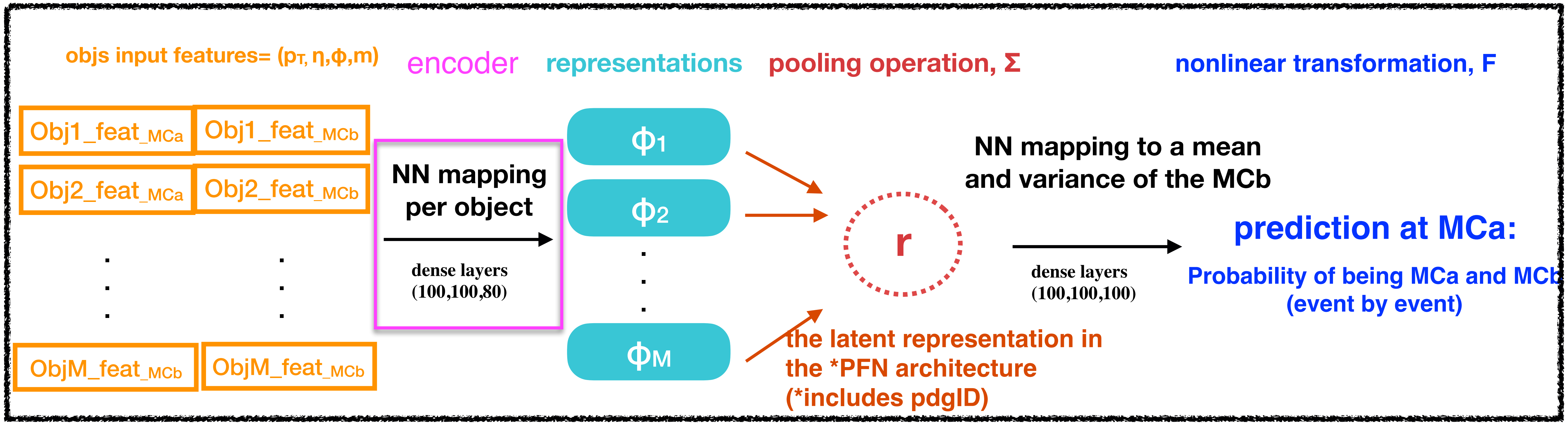
- Backgrounds: ttbar, **V+jets**, multijets, single top, Diboson.

- We aim to have a generic classification for VH(bb) as well as VH(cc) analyses in both boosted and resolved regimes.
  → generate a mapping function between two MC configurations that is independent of the reconstruction scheme.
  → We will use two dense neural networks in TensorFlow's Keras to solve the classification problem inclusively (without b-tagging).

- Computing Challenges.

Deep Set Neural Network

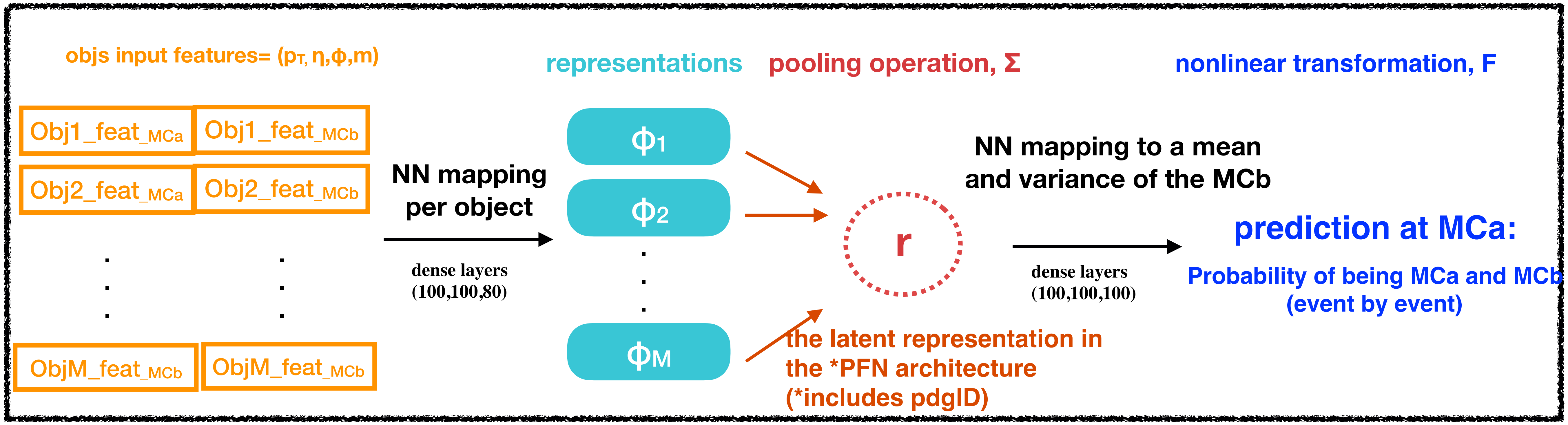
- The DSNN architecture (ref.) $\mathcal{O}(p_1,\ldots,p_M) = \mathrm{F}(\Sigma_{i=1}^{M}\Phi(p_i))$


objs input features= (p_T, η,ϕ,m)
encoder
representations
pooling operation, Σ
nonlinear transformation, F
Obj1_feat_MCa
Obj1_feat_MCb
Obj2_feat_MCa
Obj2_feat_MCb
ObjM_feat_MCb
ObjM_feat_MCb
NN mapping per object
dense layers (100,100,80)
ϕ1
ϕ2
ϕM
r
NN mapping to a mean and variance of the MCb
dense layers (100,100,100)
prediction at MCa:
Probability of being MCa and MCb
(event by event)
the latent representation in the *PFN architecture
(*includes pdgID)

- **Encoder Φ**(p_i): embed datasets into an appropriate **vector space**.
  → The approximate function will accept this input sets, which have no order.
  → The algorithm must operate on representations of sets to be invariant to their ordering (ref).
- **Decoder** F: maps the latent representation to a mean and variance for the predictive distribution.
- Vectors can be benefit from performing operations in parallel. (GPUs are needed!)

Deep Set Neural Network

- The DSNN architecture (ref.) $\mathcal{O}(p_1,...,p_M) = \mathrm{F}(\Sigma_{i=1}^{M}\Phi(p_i))$



objs input features= (p_T, η,ϕ,m)

representations

pooling operation, Σ

nonlinear transformation, F

Obj1_feat$_{\mathrm{MCa}}$ Obj1_feat$_{\mathrm{MCb}}$

Obj2_feat$_{\mathrm{MCa}}$ Obj2_feat$_{\mathrm{MCb}}$

ObjM_feat$_{\mathrm{MCb}}$ ObjM_feat$_{\mathrm{MCb}}$

NN mapping per object

dense layers (100,100,80)

ϕ_1

ϕ_2

ϕ_M

NN mapping to a mean and variance of the MCb

r

dense layers (100,100,100)

prediction at MCa:

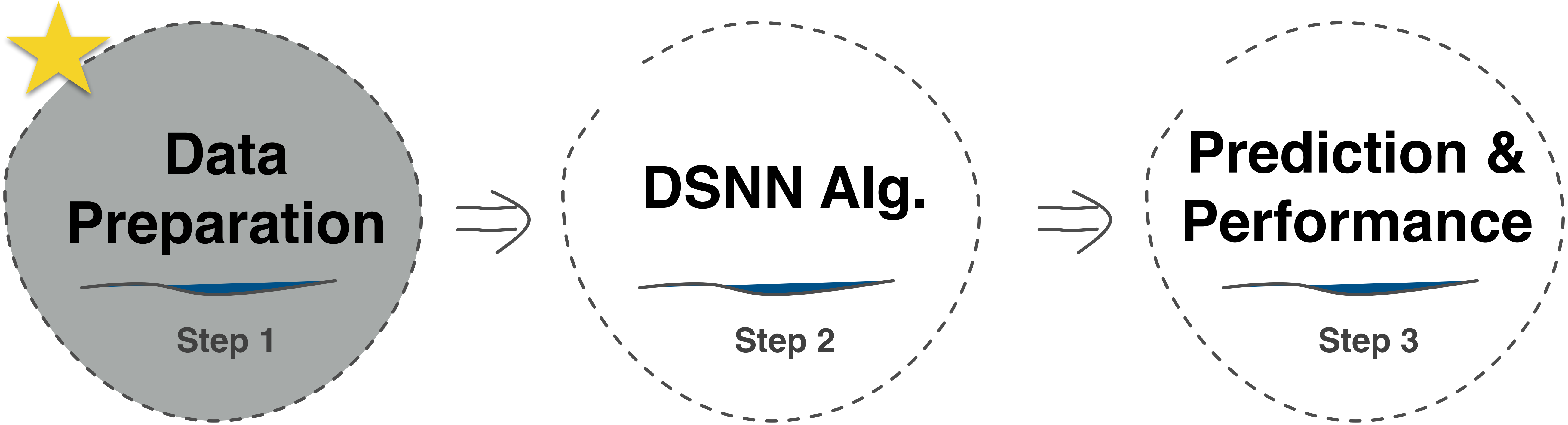
Probability of being MCa and MCb (event by event)

the latent representation in the *PFN architecture (*includes pdgID)

- Computing challenges in the DSNN:
  - High RAM and CPU usage.

    → To make MCa+d+e Numpy arrays from the CxAOD samples requires >250 GB memory!
  - Time consuming in data preprocessing and the NN training.

    → To get a trained model usually takes 5-6 hours with the full Sherpa and MGPy8 datasets.



Data Preparation

Step 1

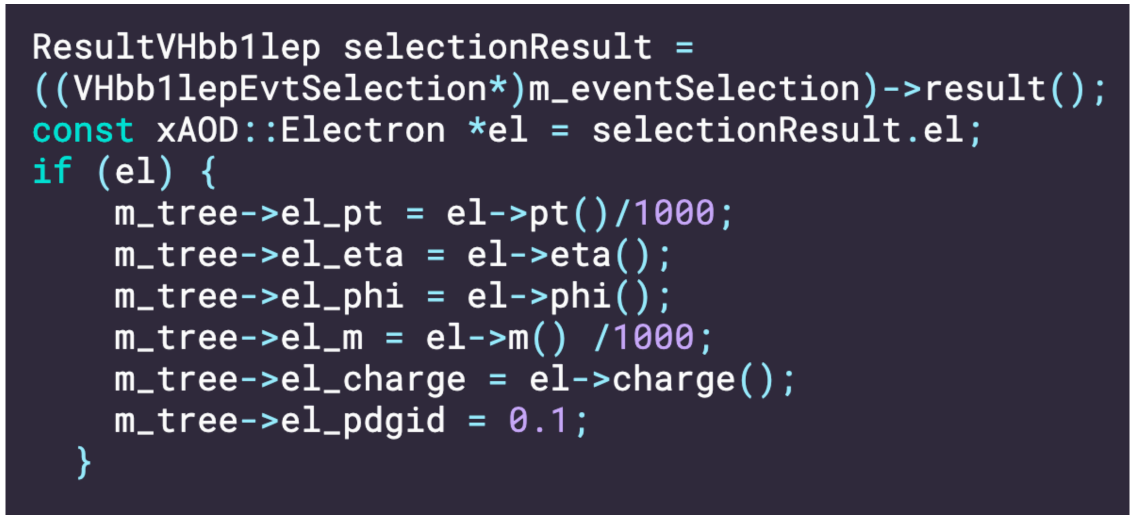
DSNN Alg.

Step 2

Prediction & Performance

Step 3

Data Preprocessing



```
ResultVHbb1lep selectionResult =
((VHbb1lepEvtSelection*)m_eventSelection)->result();
const xAOD::Electron *el = selectionResult.el;
if (el) {
    m_tree->el_pt = el->pt()/1000;
    m_tree->el_eta = el->eta();
    m_tree->el_phi = el->phi();
    m_tree->el_m = el->m() /1000;
    m_tree->el_charge = el->charge();
    m_tree->el_pdgid = 0.1;
  }
```

- Store 4-vector sets of interested objects that pass certain criteria from CxAOD.

- Convert information from CxAOD to Numpy arrays in a PFN tensor format: (events(N) x objects(6) x features(5)) dimension.

MC = [[[Pt_ele1,Eta_ele1,Phi$_1$_ele1, M$_1$_ele1, pdgID_ele1],[Pt$_1$_ µ1,Eta_µ1,Phi_µ1, M_µ1, pdgID_µ1],…,
[Pt_met1,Eta_met1,Phi_met1, M_met1, pdgID_met1]

.

.

.

[[Pt_ele$_N$,Eta_ele$_N$,Phi$_1$_ele$_N$, M$_1$_ele$_N$, pdgID_ele$_N$],[Pt$_1$_ µ$_N$,Eta_µ$_N$,Phi_µ$_N$, M_µ$_N$, pdgID_µ$_N$]…,
[Pt_met$_N$,Eta_met$_N$,Phi_met$_N$, M_met$_N$, pdgID_met$_N$]

EventWeight = [Weight$_1$, Weight$_2$,….. Weight$_N$]

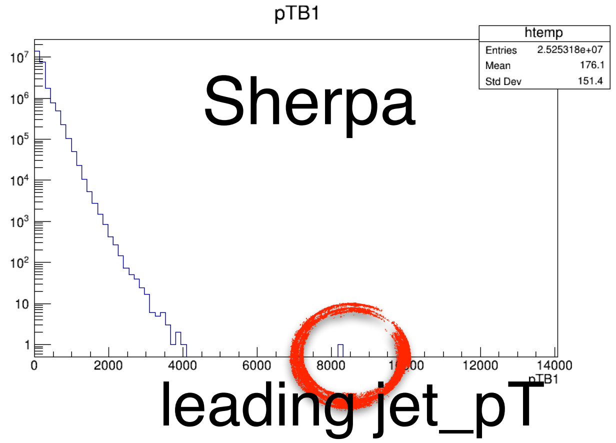
- It's not possible to run all pileup campaigns together, so we must run them 3 times separately to get Numpy first.

	Sherpa (events)	MGPy8 (events)	Time for getting numpy arrays	MaxRSS
MCa	25053101	7570460	02:32:43	27 GB
MCd	30415633	9017054	02:54:44	139 GB
MCe	40936533	11854840	05:06:13	186 GB
Total	96,405,267	28,442,354	x	>250 GB

Data Scaling

- Reasons for scaling:
  - To prevent one significant number from playing a decisive role because of the magnitude (at pooling operation stage).
  - To get faster neural network gradient descent converge.

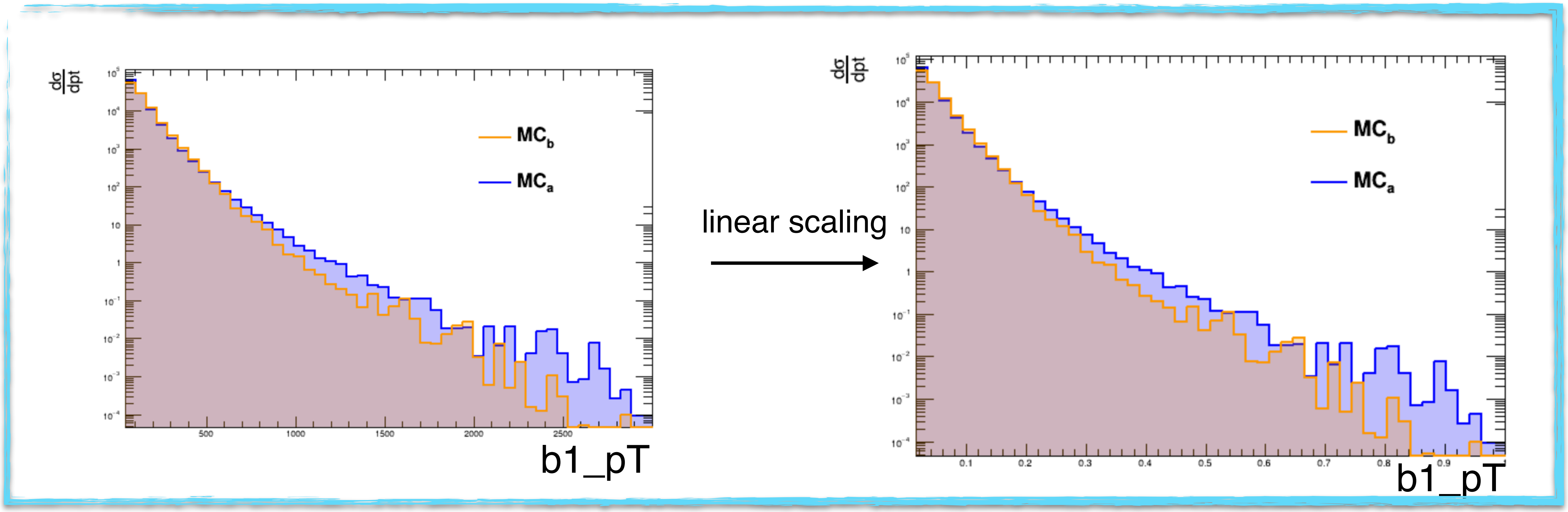
$$X' = \frac{x - x_{min}}{x_{max} - x_{min}}$$



- The feature scaling technique: Normalization.

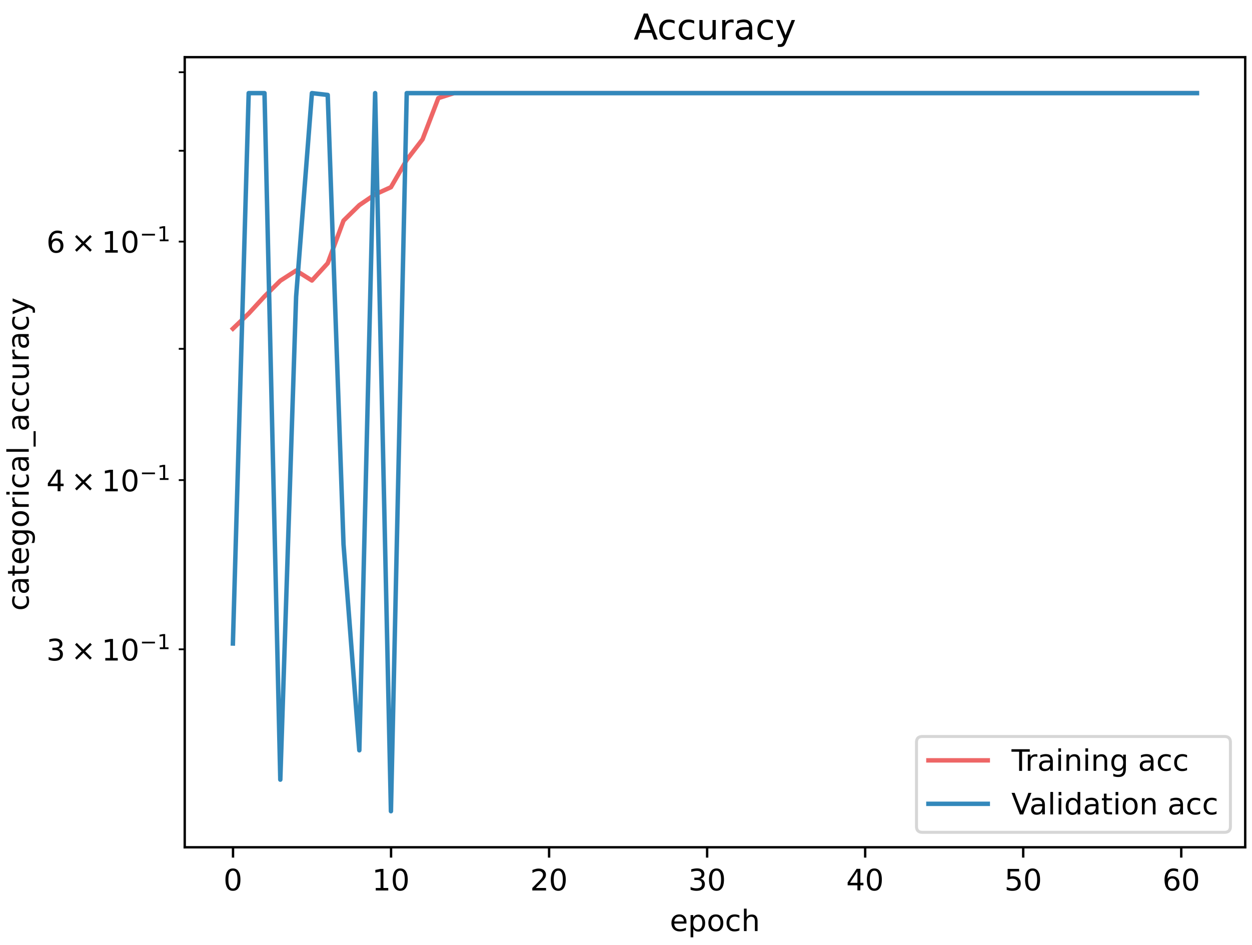
⭐ Remove outliers; e.g. pT > 3000 GeV

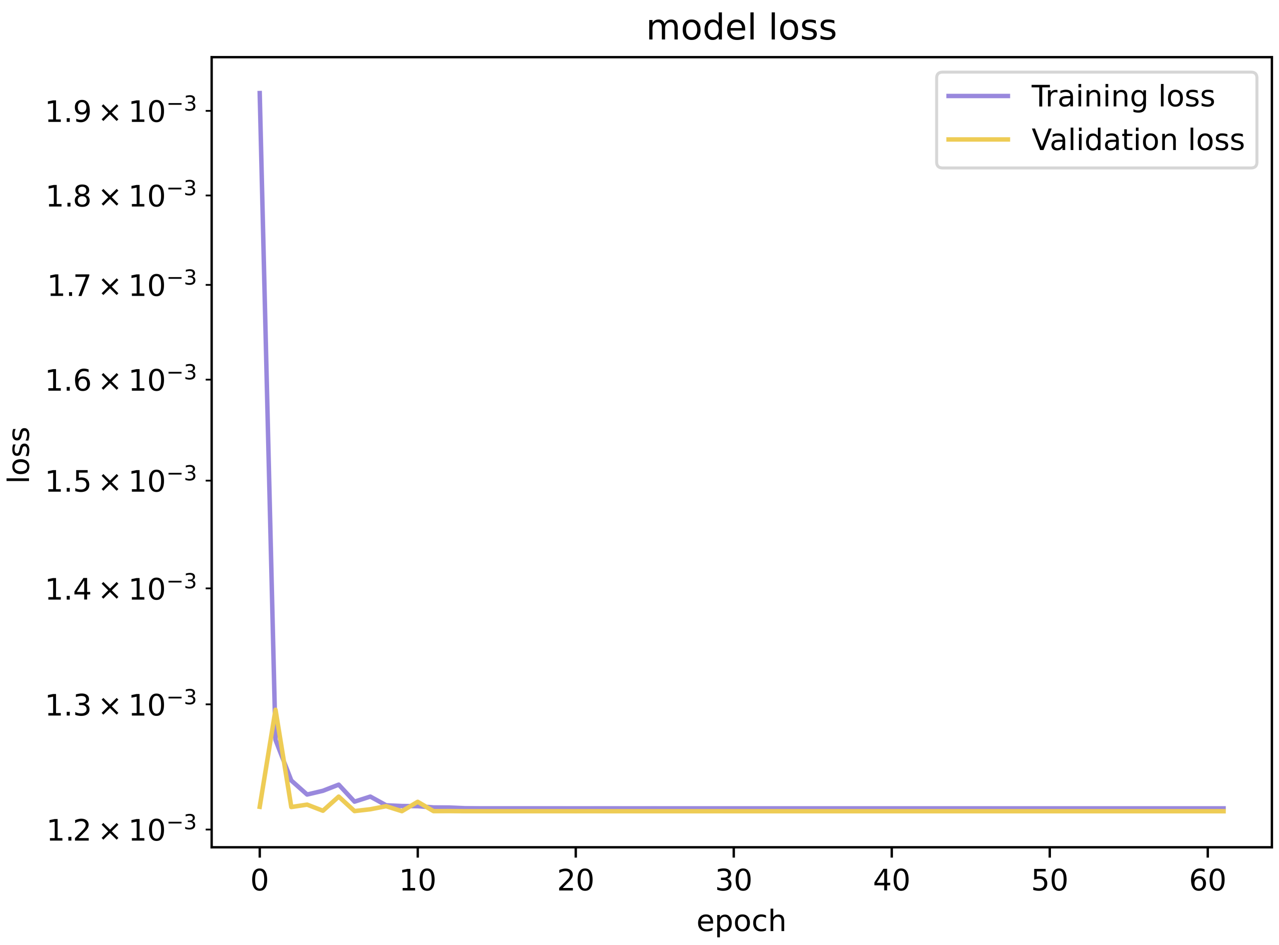
⭐ Rescale the Sherpa and MGPy8 by the max and min features of the Sherpa + MGPy8.



Outliers Matter

- Machine learning algorithms are sensitive to the range and distribution of attribute values. Data outliers can spoil and mislead the training process resulting in longer training times, less accurate models and ultimately poorer results. (ref)
- To remove or not to remove?
  - Normalized scaling is sensitive to outliers.
  - Removing > 3TeV jets for now is acceptable for a comparison of methods.



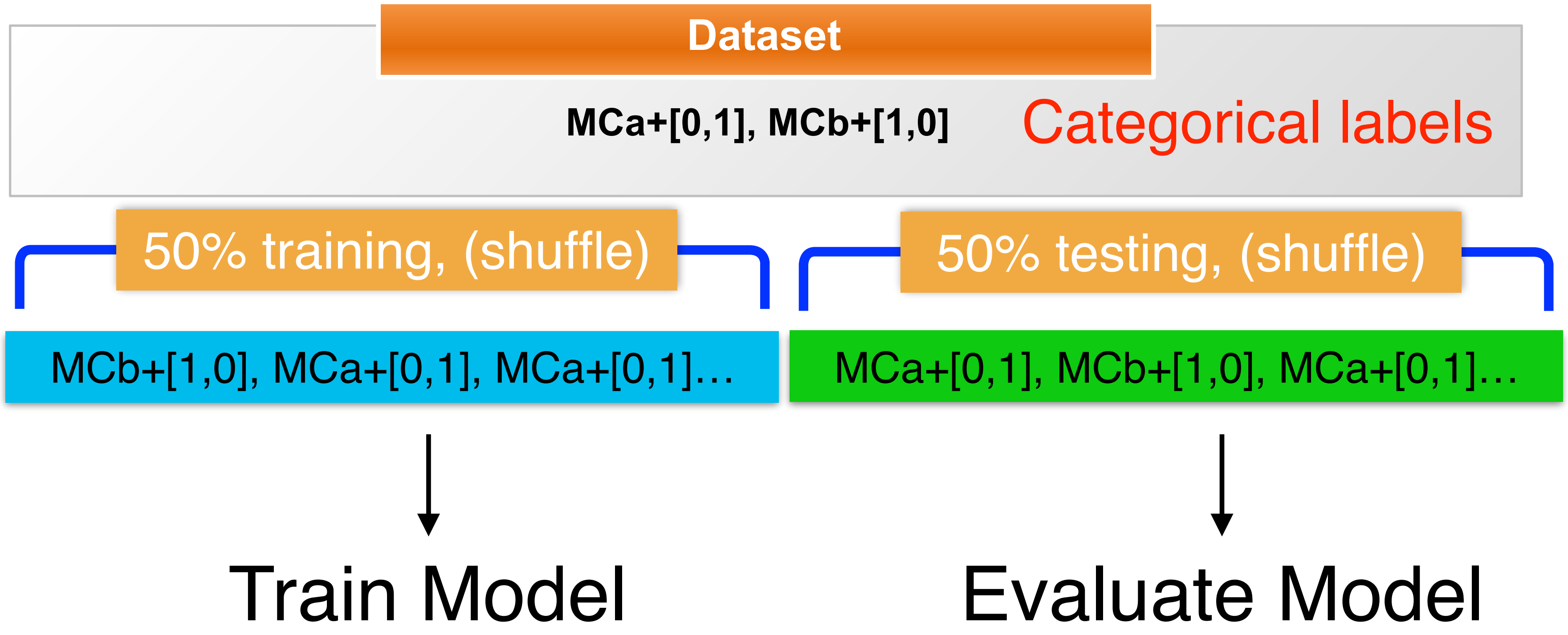


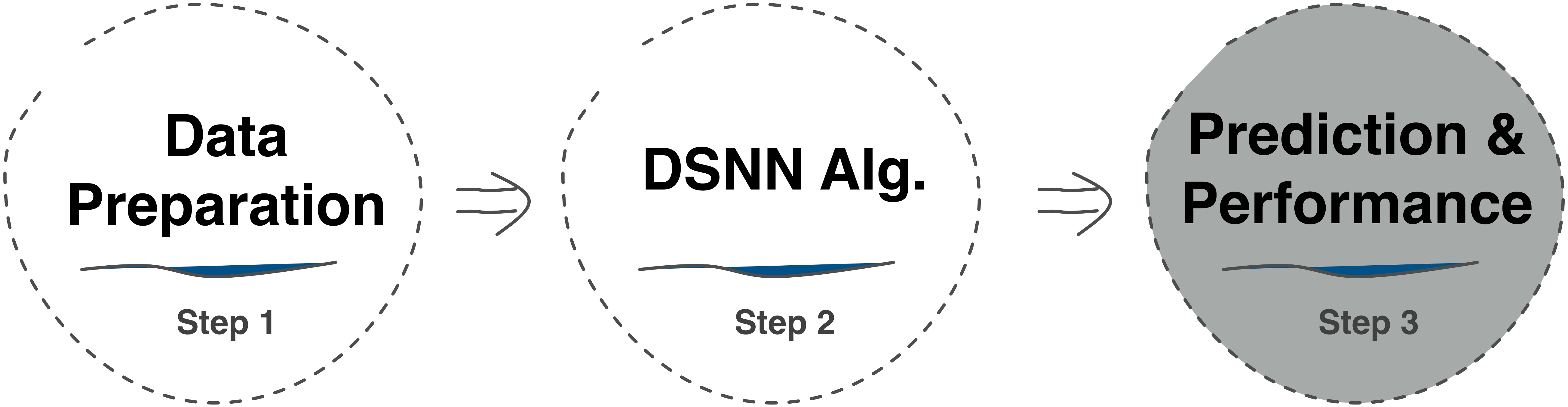
Train & Test

- A training dataset: feed into the model so that the NN can learn.
- A testing dataset: use to evaluate the model.
    - test the model on data that we have never used/seen.
- The NN learns in a supervised way.
    - Both training and testing datasets are assigned categorical labels [0,1] and [1,0] representing Sherpa and MGPy8.

→ One-hot encoded doesn't have any ranking for category values.

→ Easy to determine a prior probability.





Data Preparation

Step 1

DSNN Alg.

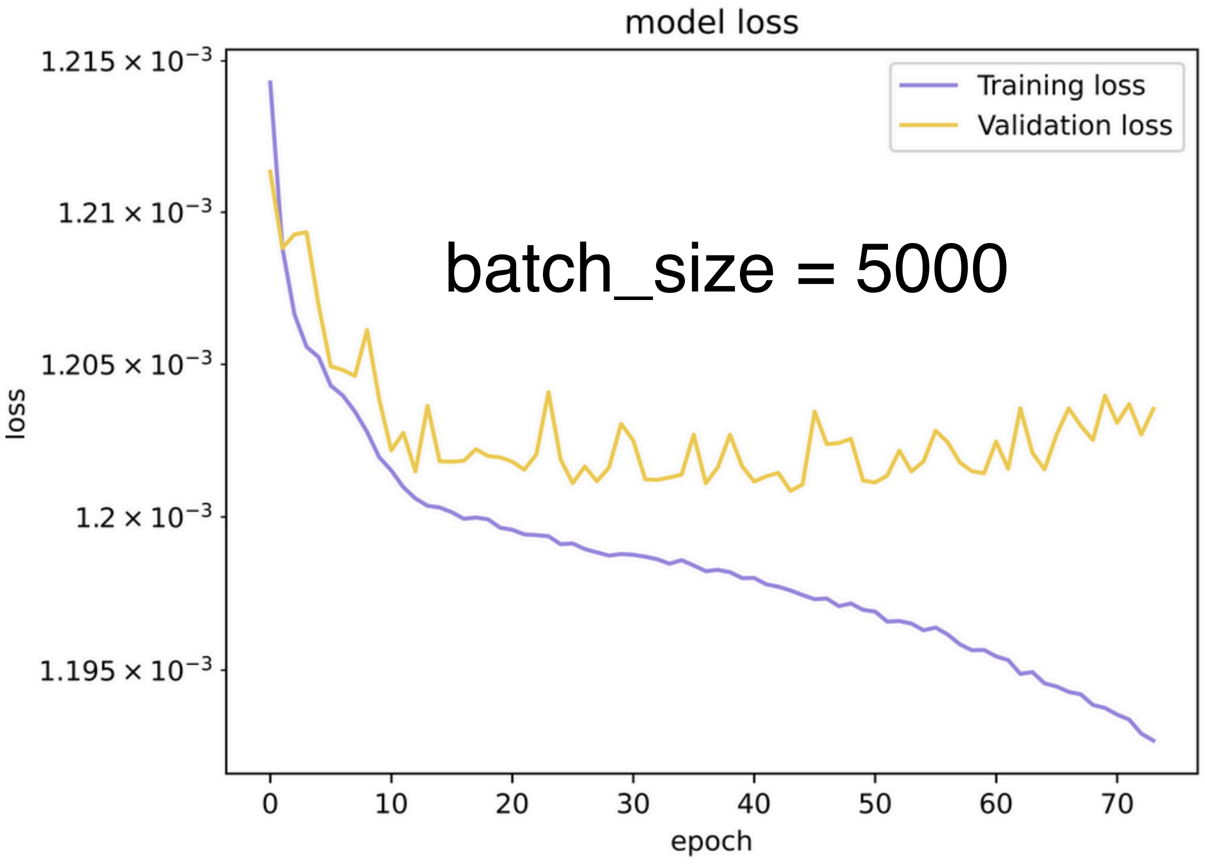
Step 2

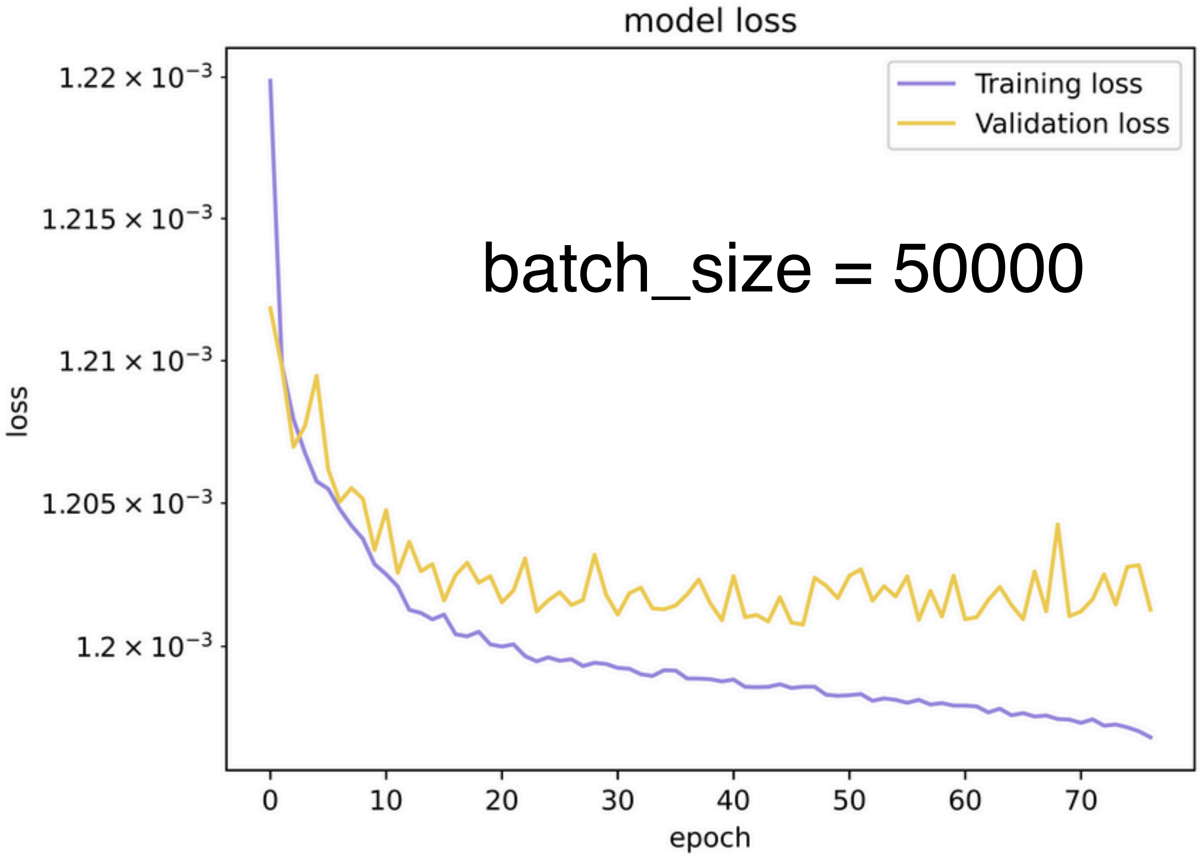
Prediction & Performance

Step 3

Categorical Cross Entropy

- Hyper-parameters: nodes, layers, batch_size… etc.
- Changing batch_size = 50000 (to make sure the model can walk through enough data before updating its parameters.)
- ❌ Unrepresentative training dataset:

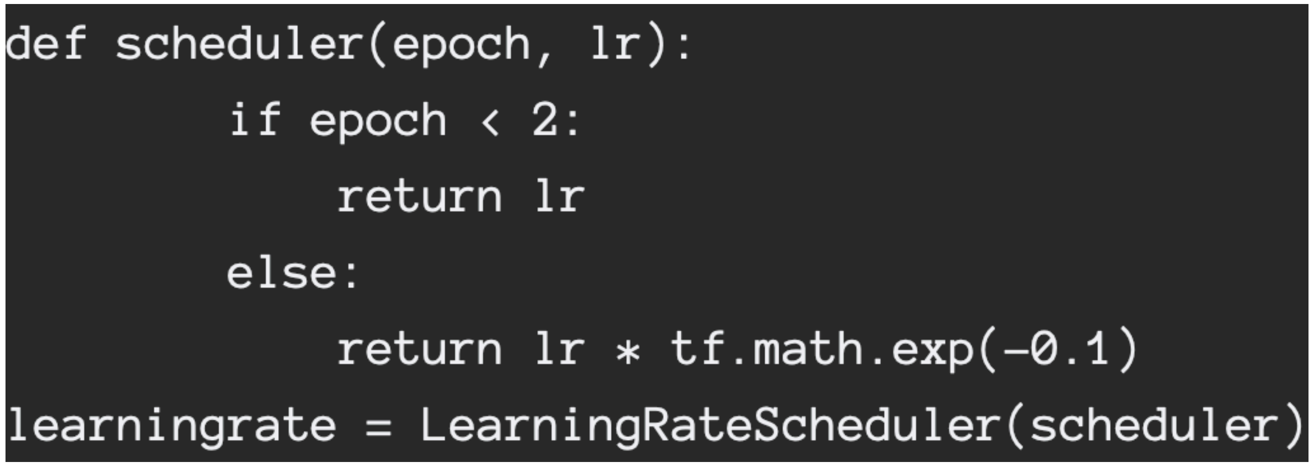




- The gap between two learning curves was reduced through adding more datasets to mini batches.
- Despite adding more datasets in (have tried batch_size=100000), the NN is not able to capture characteristics on training dataset well.

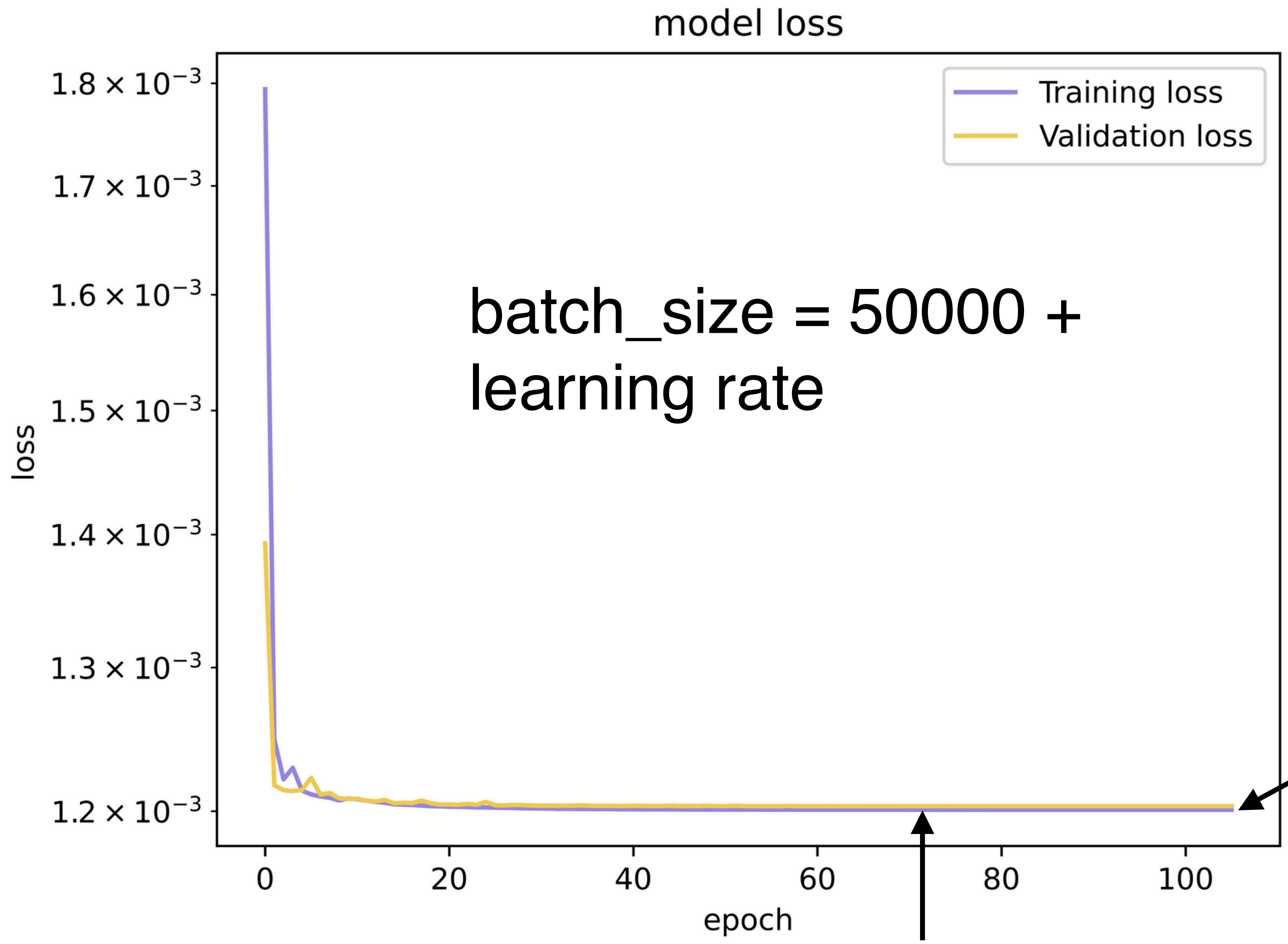
Categorical Cross Entropy

- Optimizer: Adam
- Get an updated learning rate!



```
def scheduler(epoch, lr):
        if epoch < 2:
            return lr
        else:
            return lr * tf.math.exp(-0.1)
learningrate = LearningRateScheduler(scheduler)
```

```
>>> # This function keeps the initial learning
rate for the first 2 epochs and decreases it
exponentially after that. (ref)
```



batch_size = 50000 + learning rate

a generalization gap, good fit!

the 76th got the minimal loss

Probability

- The final NN layer returns the raw values for the predictions (= logits).
- Softmax is used as a default recommended activation function.
  - Func$_{softmax}$ (logits) => Probability for each class.

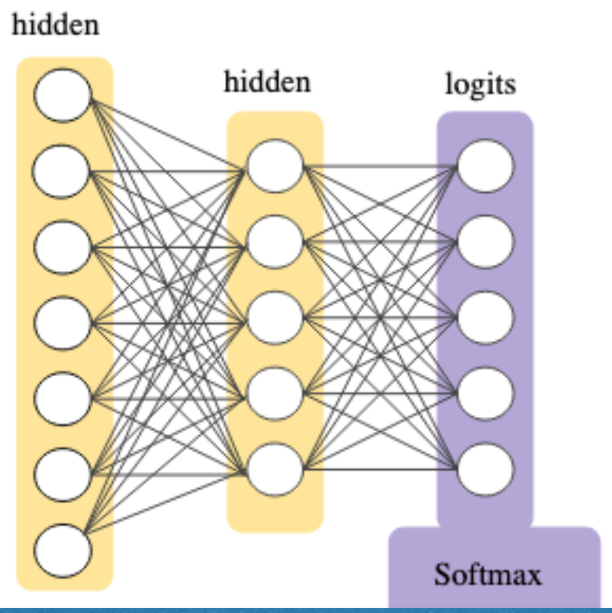
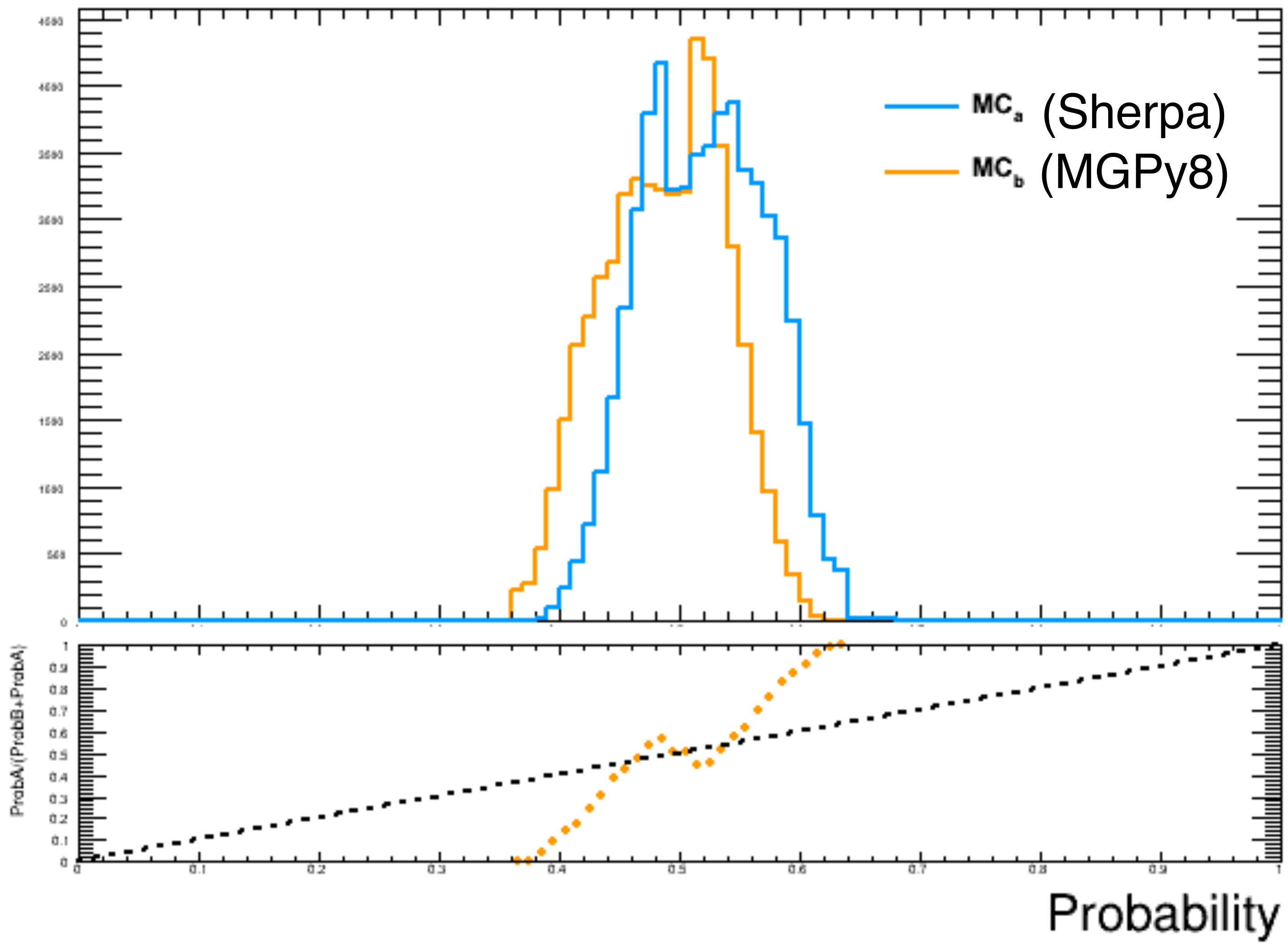
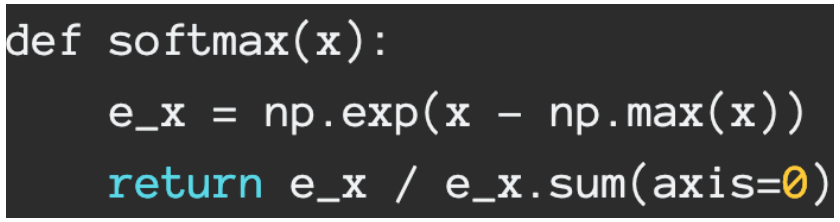


fig. source here
Common DSNN architectures options, here.





```
def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0)
```

Categorical Accuracy

- The final NN layer returns the raw values for the predictions (= logits).
- Softmax is used as a default recommended activation function.
  - Func$_{softmax}$ (logits) => Probability for each class.

$$Y_{True} = \begin{bmatrix} [0,1] \\ [0,1] \\ [1,0] \end{bmatrix} \xrightarrow{\text{argmax()}} Y_{True} = [1,1,0]$$

$$Y_{Pred} = \begin{bmatrix} [0.45, 0.55] \\ [0.65, 0.35] \\ [0.53, 0.47] \end{bmatrix} \xrightarrow{\text{argmax()}} Y_{Pred} = [1,0,0]$$

source code, here.

- Bad model can possibly end up with high accuracy.

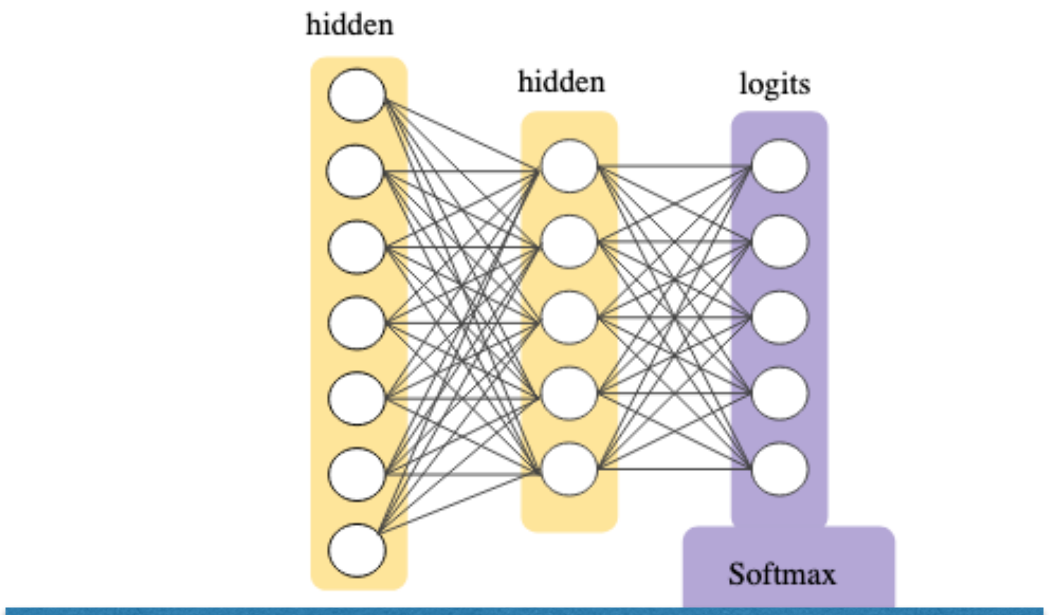
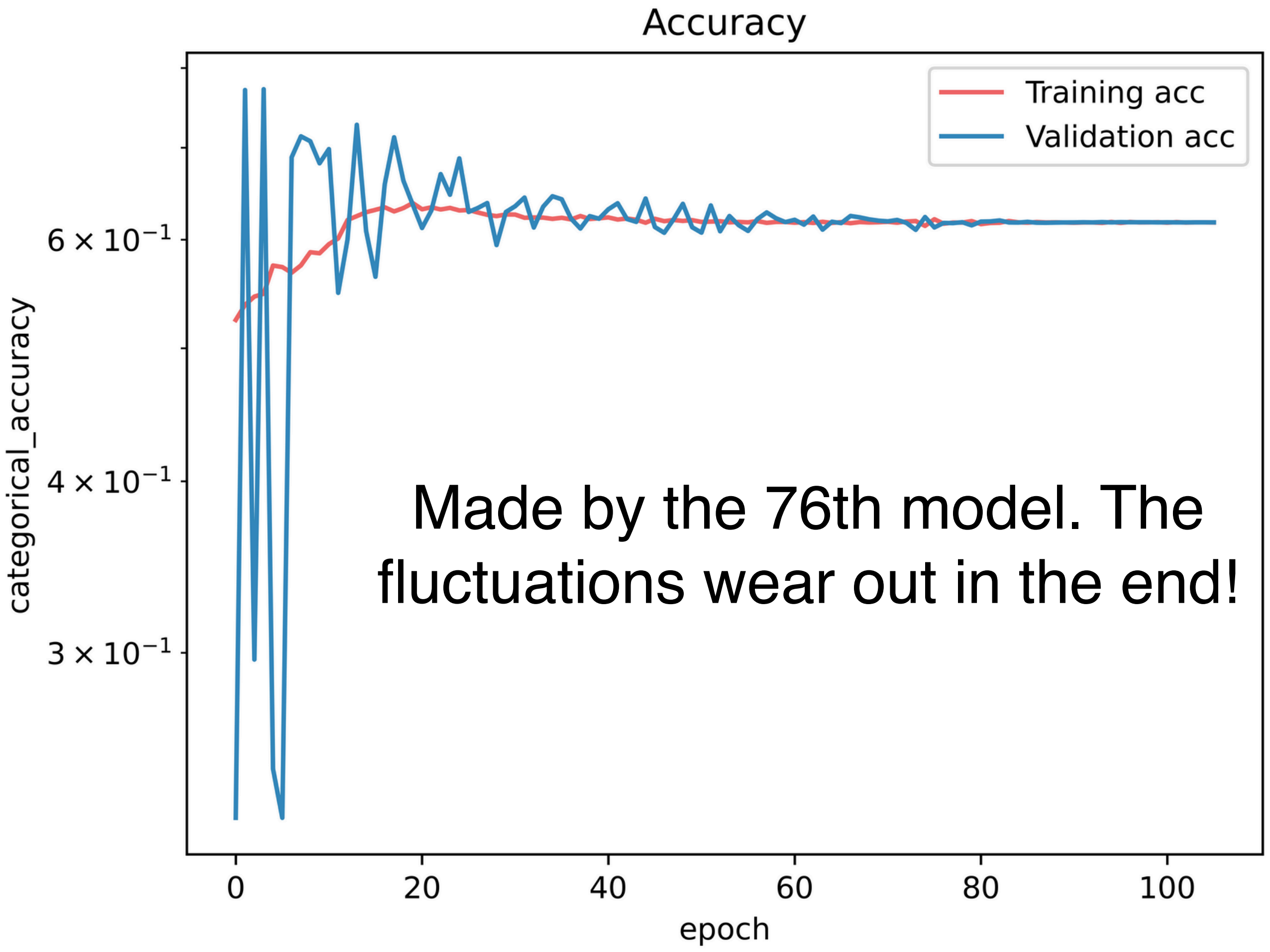


fig. source here
Common DSNN architectures options, here.



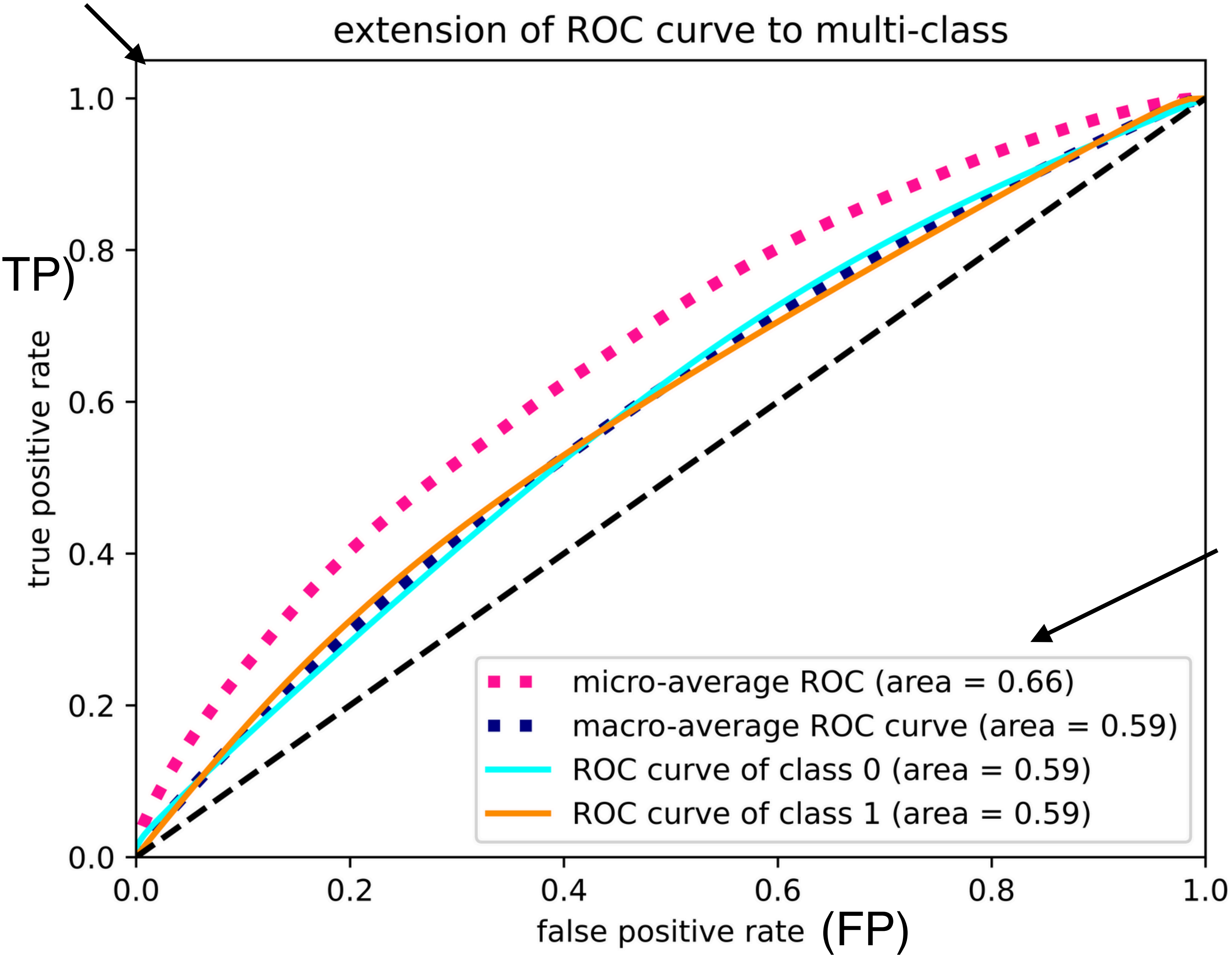
Made by the 76th model. The fluctuations wear out in the end!

ROC

- The ROC was made by the 76th model.

		True Class	
		Sherpa	MGPy8
Predicted Class	sherpa	TP	FP
	MGPy8	FN	TN

the MCa are classified correctly
the MCb are classified correctly


extension of ROC curve to multi-class

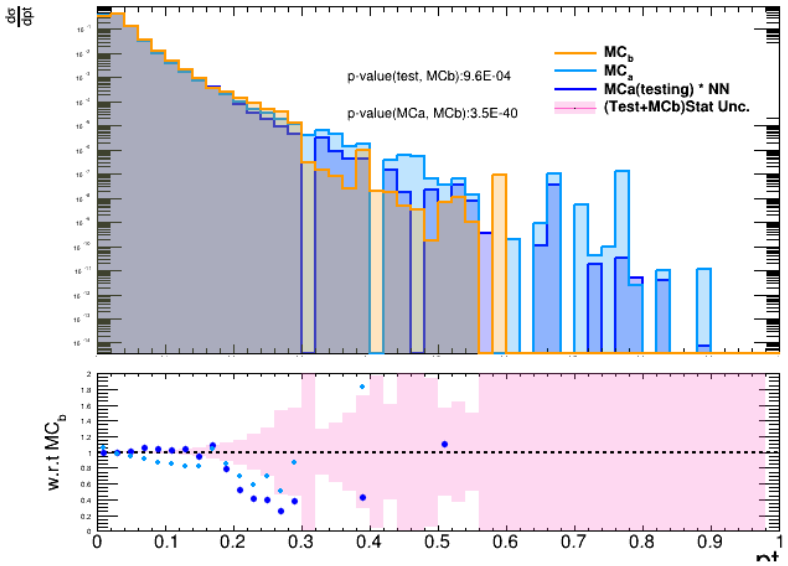
(TP)

(FP)

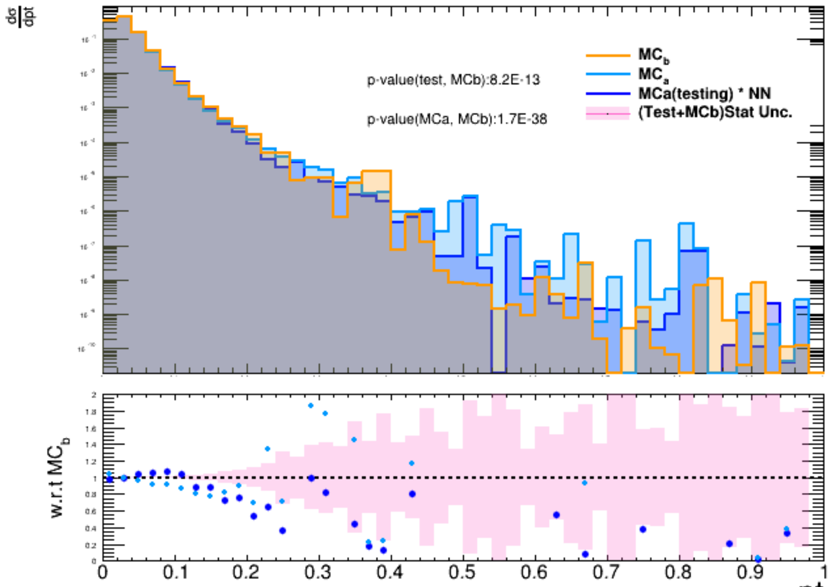
Interpretation of area under the curve(AUC)
AUC > 0.5: the classifier is able to detect a higher number of TP and TN than FP and FN.

Features Performance

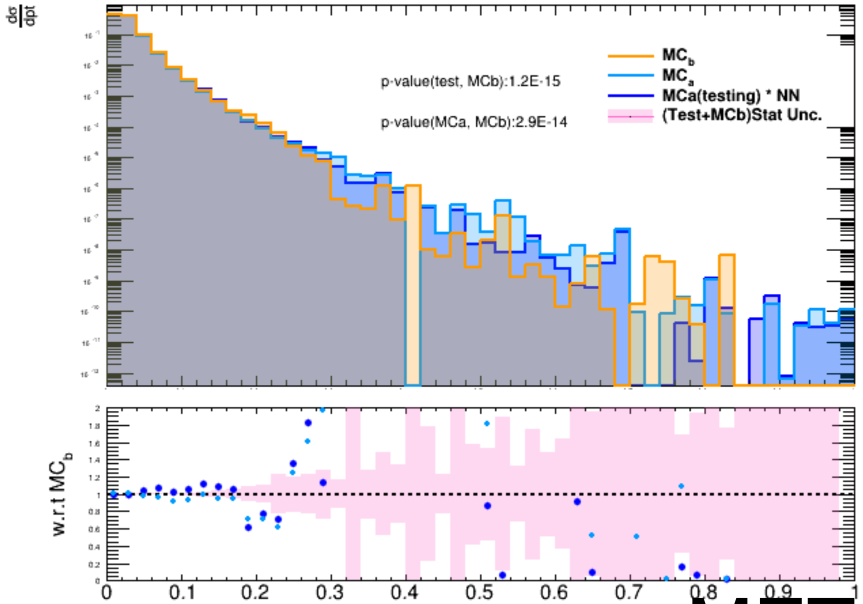
- Sherpa * NN Weight → MGPy8.
- NN Weight = Prob(MGPy8)/Prob(Sherpa). MCa= Sherpa; MCb=MGPy8



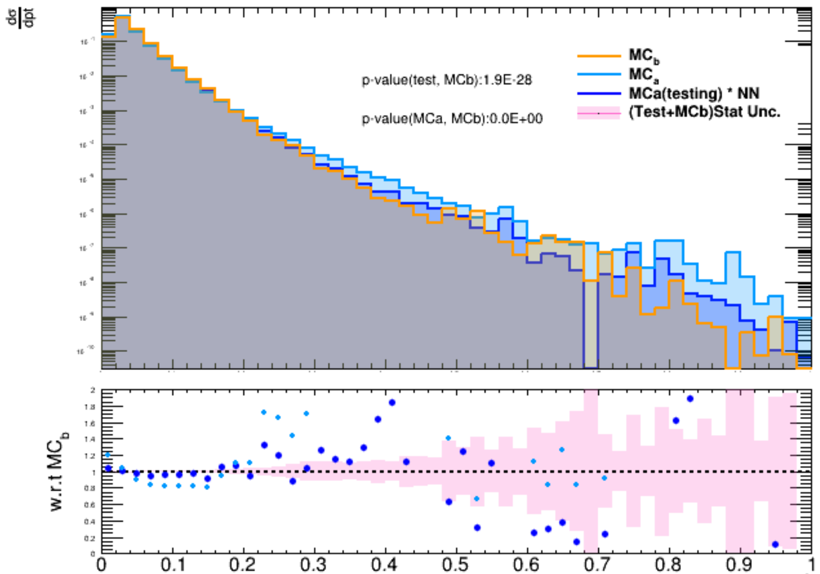
electron pT



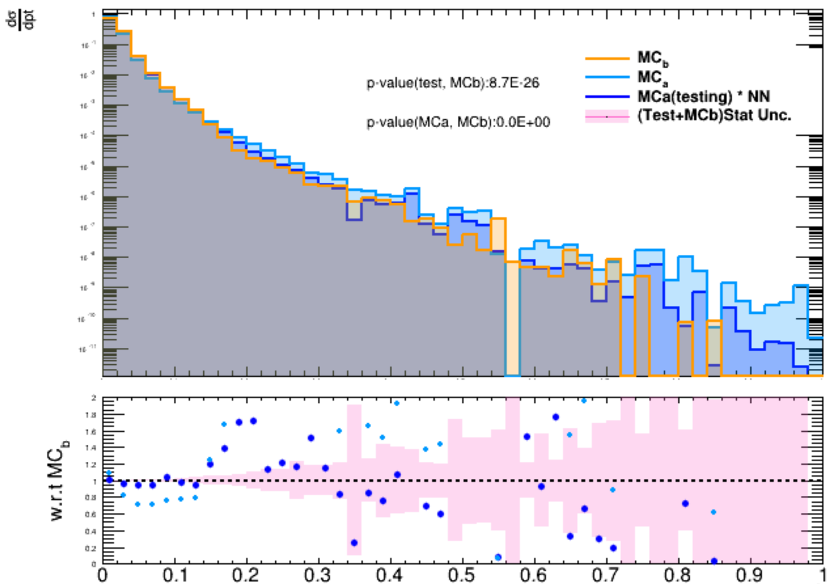
muon pT



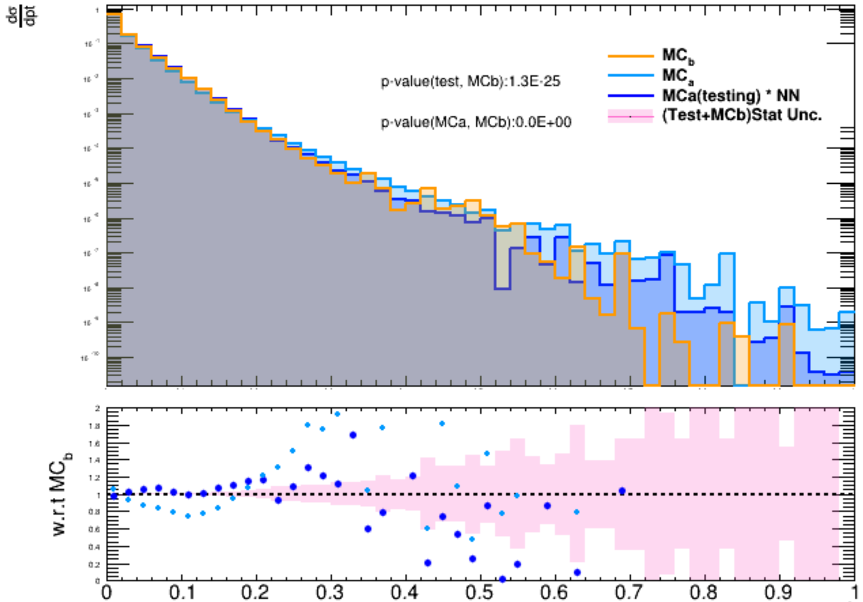
MET



leading jet pT



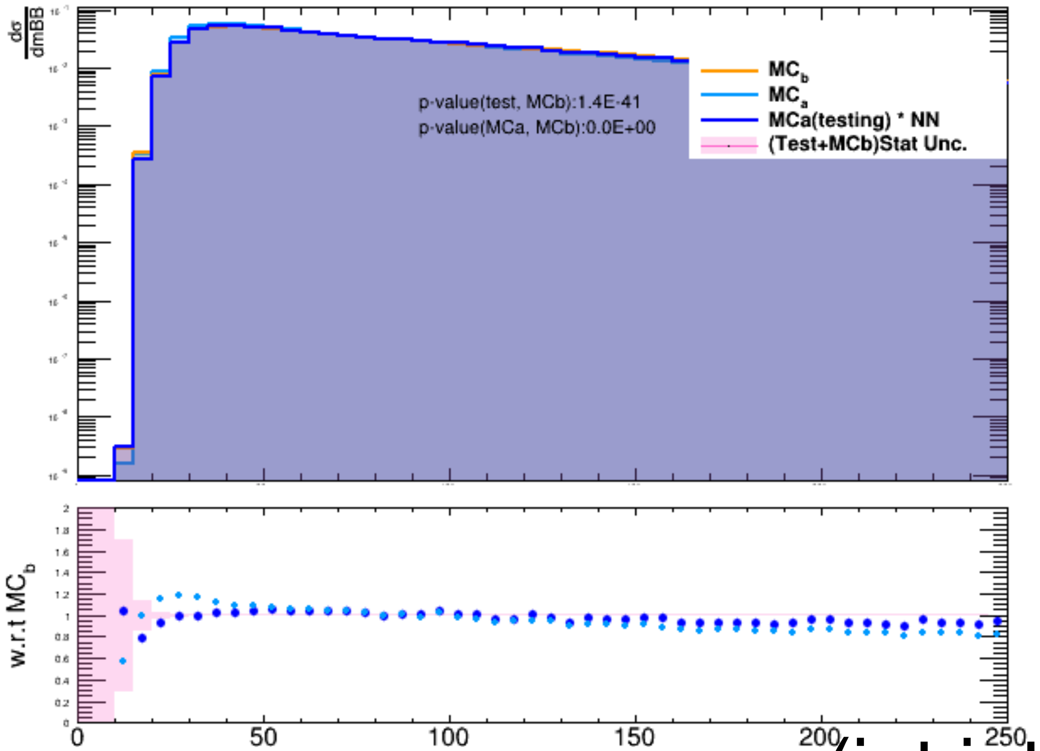
sub-leading jet pT



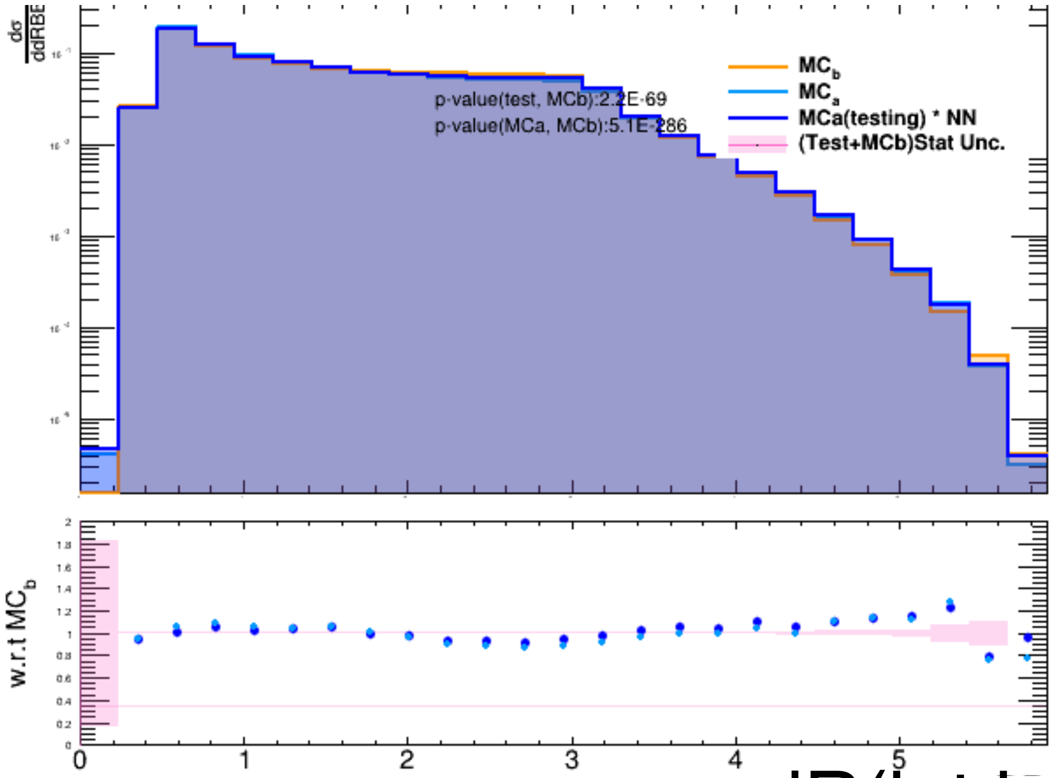
3rd jet pT

Observables Performance

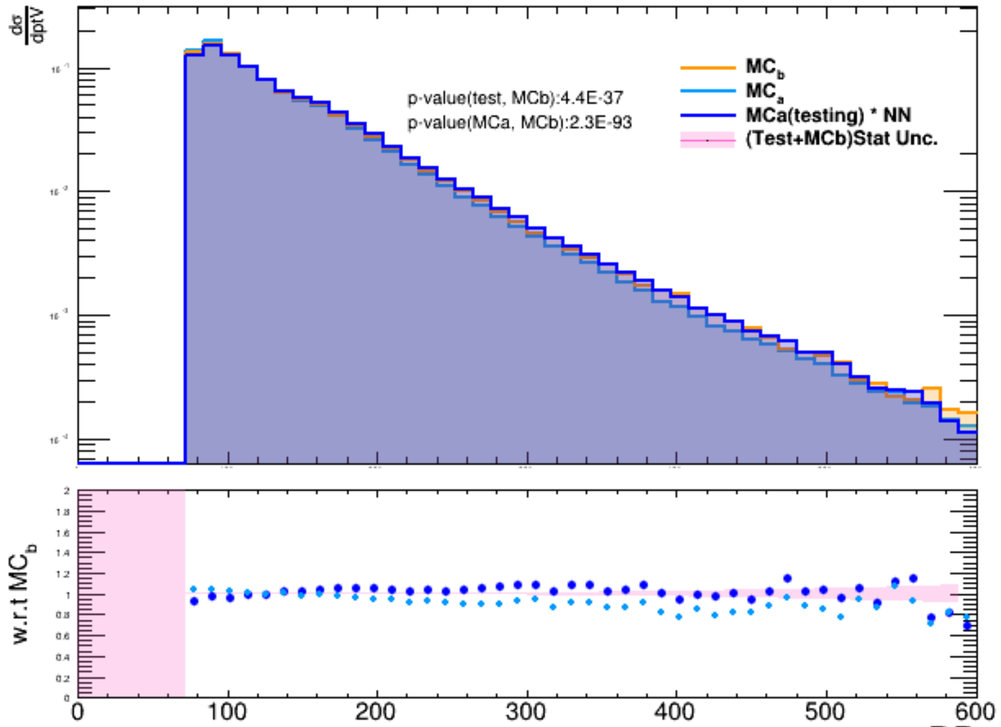
- Use spectator (mass(j,j), ptV, dR(j,j)…) variables stored in the ntuples * NN Weights. (50% of the Sherpa+MGPy8 samples on testing.)



m(jet,jet)



dR(jet,jet)



ptV

Computing tactics

- Speeding up data pre-processing using DASK on GCP.
  - The parallel data processing can be done in 5-6 mins (the final amount of memory occupied ~ 47.4GB), while the standard data processing takes at least 40 mins.

- Speeding up hyper parameter tuning using Ray APIs on NERSC computing facilities.
  - I asked for 4 nodes, 32 tasks, and 32 GPUs to test the parallel training with full Sherpa+MGPy8 datasets. To scan 4 combination of hyper-parameters, the job was done in 54.7 mins using Ray clusters, while w/o distributed computing, the same size of inputs was done in 87.8 mins.
  - Failed to use DASK to train TF with larger datasets in parallel on GCP.

- Distributed tasks using a user container through ATLAS PanDA, prun.

```
prun - - containerImage docker://sjiggins/tensorflow-gpu-dsnnr:v1 \
--exec="./myDSNNr_run.sh '%IN' '%IN2'"\
--inDS user.sjiggins.mc15_13TeV.410470.PhPy8EG_A14_ttbar_hdamp258p75_nonallhad.evgen.EVNT.e6337.VHbb_DSNNr_ttbar-v3_1_Lep\
--secondaryDSs IN2:1:user.sjiggins.mc15_13TeV.410464.aMcAtNloPy8EvtGen_ttbar_noShWe_SingleLep.evgen.EVNT.e6762.VHbb_DSNNr_ttbar-v3_1_Lep\
--excludeFile data \
--site=GOOGLE100 \
--destSE=GOOGLE_EU\
--nFiles 1\
--outDS user.fatsai.DSNNr_output\
--outputs “GoogleCloudJob/“
```

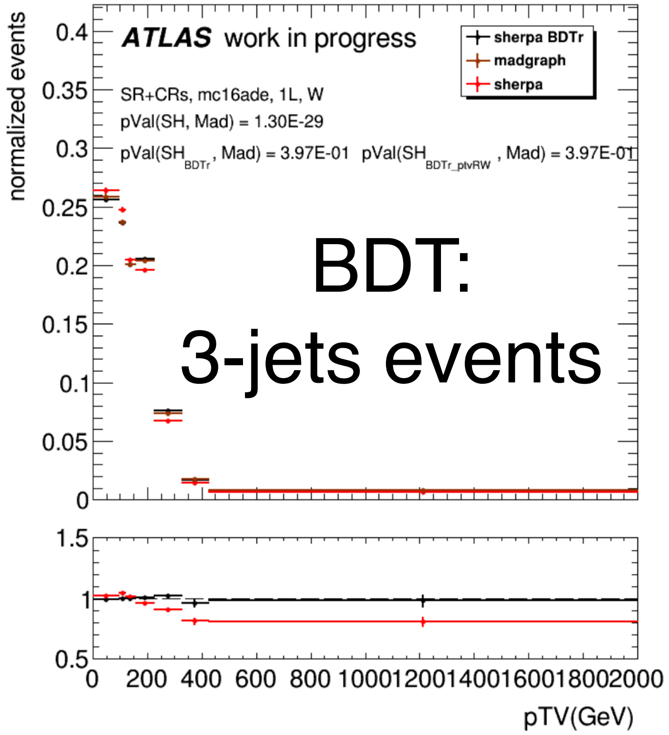
Conclusions & Plans

Conclusions:

- Good performance for classifying the inclusive events using the DSNN.
- Perhaps can do a probability calibration and see how much is improved.
- Optimize hyper-parameters could probably help get better performance.

Plans:

- Compare it with the BDT technique. (Work in progress!)
  The BDT training on the same sample is ready (Thanks to Ilaria), and next step will be making a comparison.



BDT:
3-jets events

- Working on the GCP stuff (maybe) after the LLWI2022 in Feb.

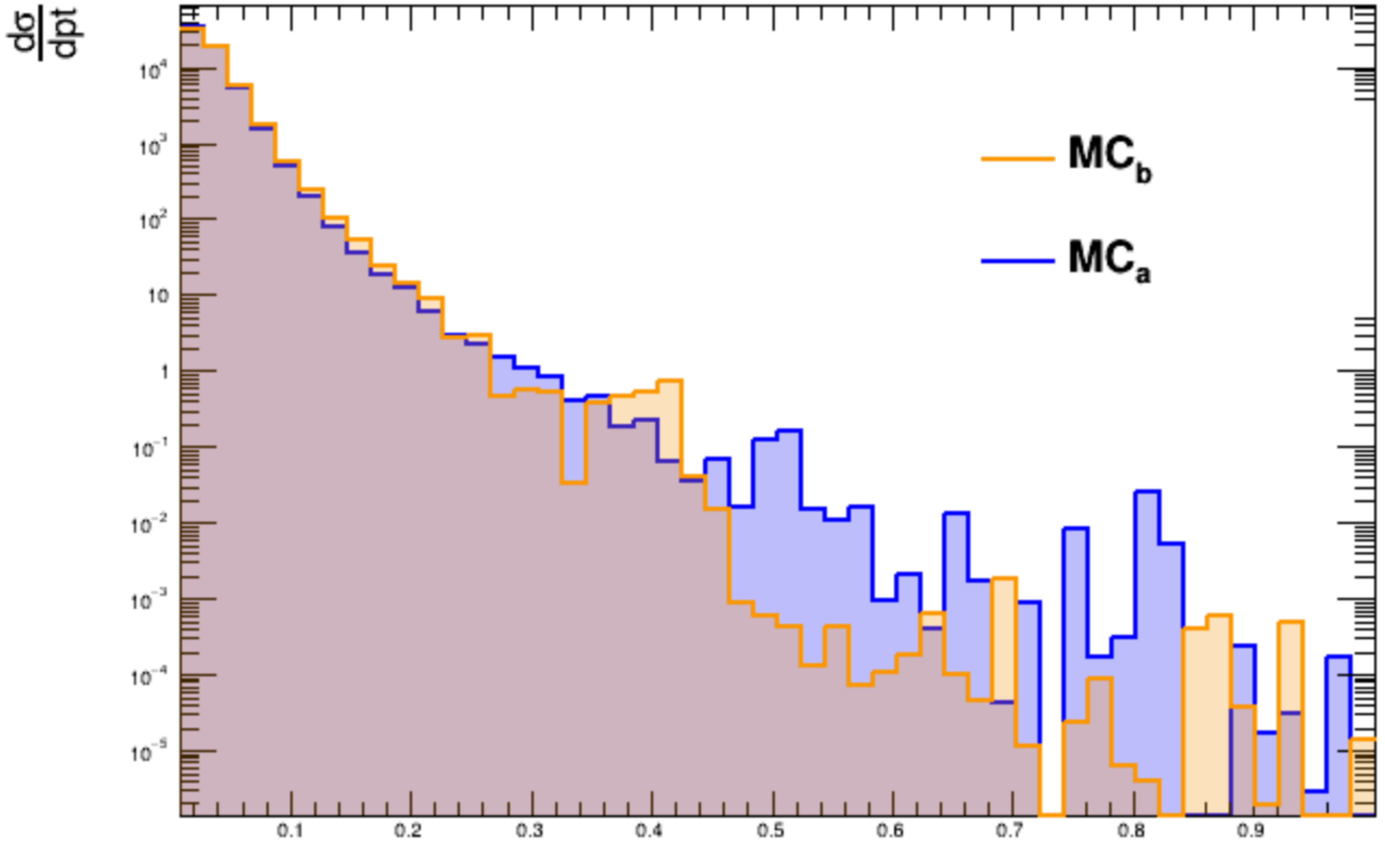
Computing needs:

- I will appreciate if I can convert three pile-up profile MCs from CxAOD to Numpy arrays in one go.
- Train the NN in parallel (DASK? Ray… others?).

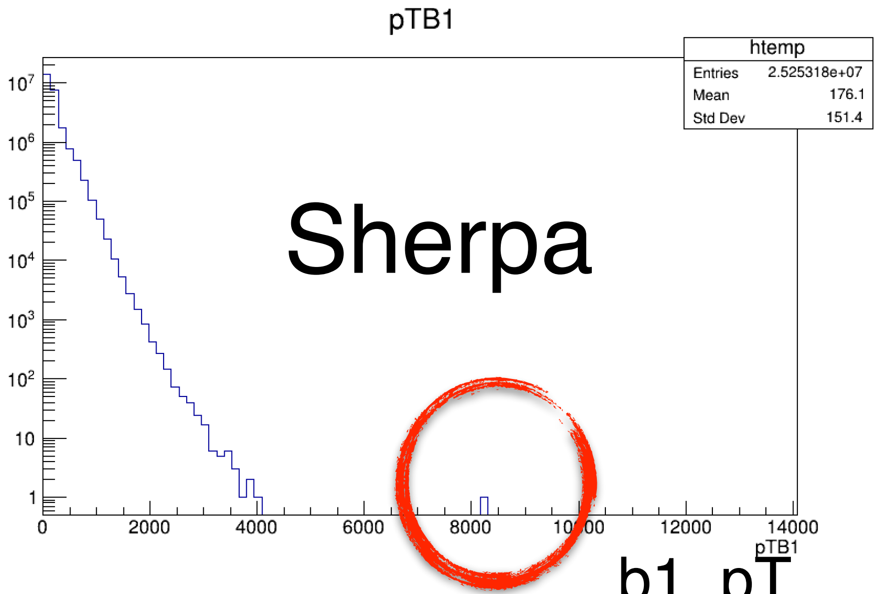
Backups

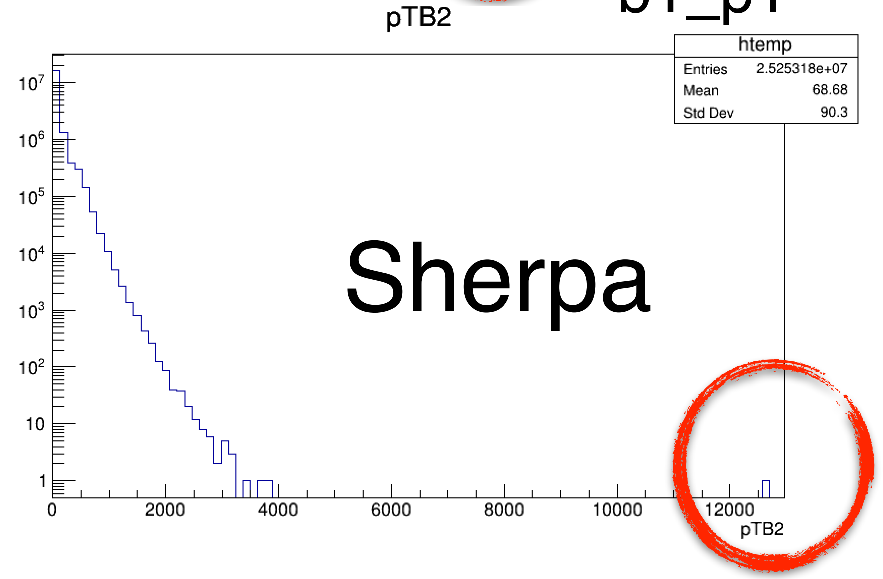
Data Scaling

- ☆ Remove outliers; e.g. pT > 3000 GeV
- ☆ Rescale the MCa by the max and min features of the MCa, and rescale the MCb by the max and min features of the MCb.



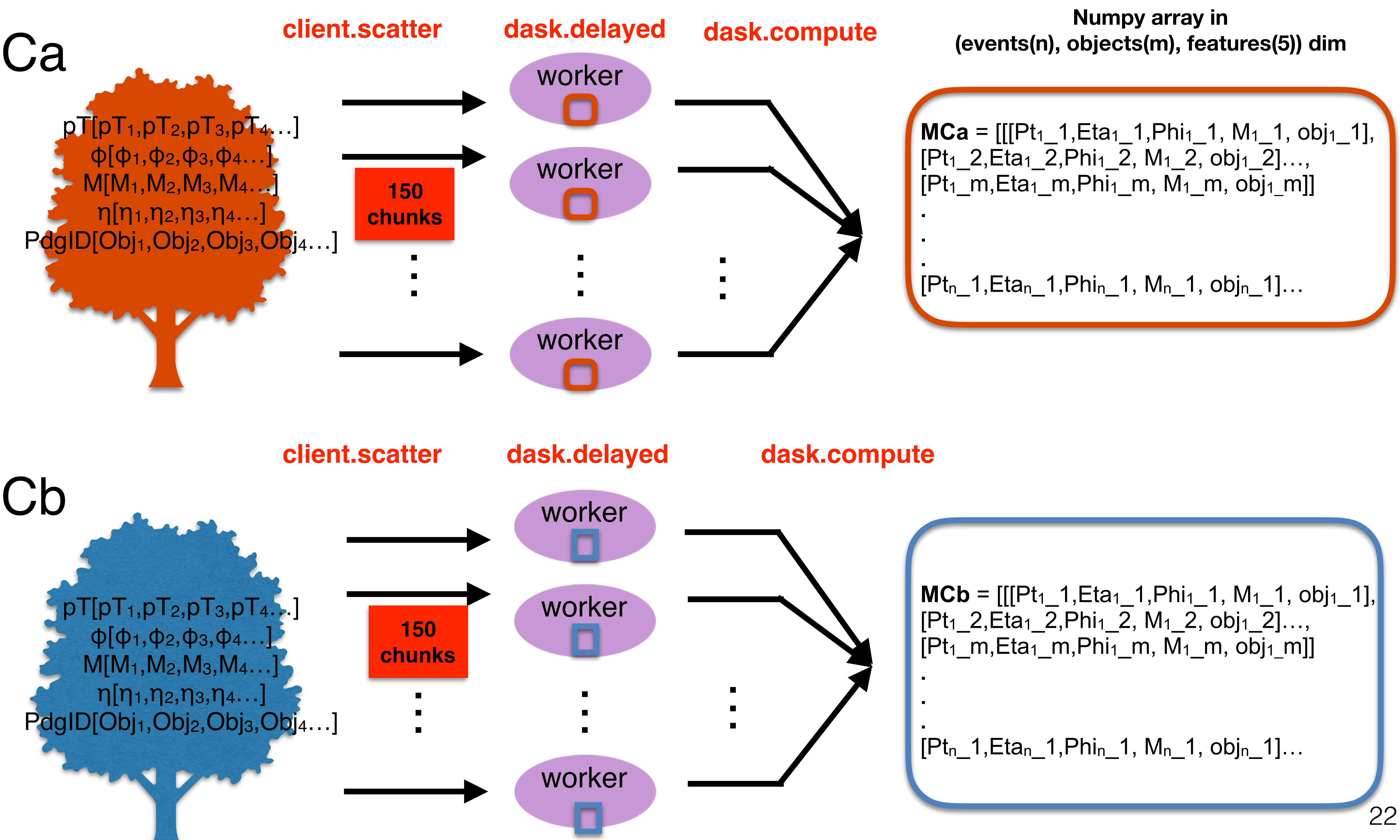
b1_pT





b2_pT

Parallel programming in data preprocessing



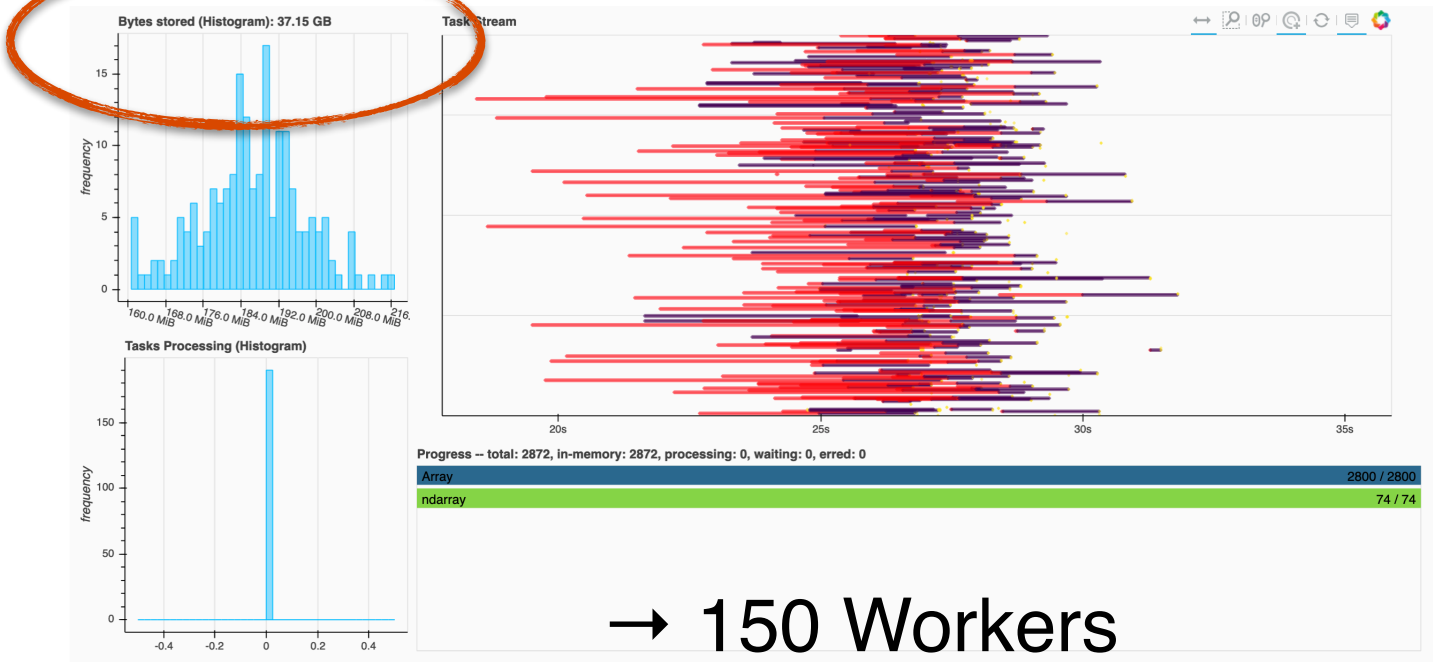
MCa
client.scatter
dask.delayed
dask.compute
Numpy array in
(events(n), objects(m), features(5)) dim
pT[pT$_1$,pT$_2$,pT$_3$,pT$_4$…]
ϕ[ϕ$_1$,ϕ$_2$,ϕ$_3$,ϕ$_4$…]
M[M$_1$,M$_2$,M$_3$,M$_4$…]
η[η$_1$,η$_2$,η$_3$,η$_4$…]
PdgID[Obj$_1$,Obj$_2$,Obj$_3$,Obj$_4$…]
150
chunks
worker
worker
worker

MCa = [[[Pt$_1$_1,Eta$_1$_1,Phi$_1$_1, M$_1$_1, obj$_1$_1],
[Pt$_1$_2,Eta$_1$_2,Phi$_1$_2, M$_1$_2, obj$_1$_2]…,
[Pt$_1$_m,Eta$_1$_m,Phi$_1$_m, M$_1$_m, obj$_1$_m]]
.
.
.
[Pt$_n$_1,Eta$_n$_1,Phi$_n$_1, M$_n$_1, obj$_n$_1]…

MCb
client.scatter
dask.delayed
dask.compute
pT[pT$_1$,pT$_2$,pT$_3$,pT$_4$…]
ϕ[ϕ$_1$,ϕ$_2$,ϕ$_3$,ϕ$_4$…]
M[M$_1$,M$_2$,M$_3$,M$_4$…]
η[η$_1$,η$_2$,η$_3$,η$_4$…]
PdgID[Obj$_1$,Obj$_2$,Obj$_3$,Obj$_4$…]
150
chunks
worker
worker
worker

MCb = [[[Pt$_1$_1,Eta$_1$_1,Phi$_1$_1, M$_1$_1, obj$_1$_1],
[Pt$_1$_2,Eta$_1$_2,Phi$_1$_2, M$_1$_2, obj$_1$_2]…,
[Pt$_1$_m,Eta$_1$_m,Phi$_1$_m, M$_1$_m, obj$_1$_m]]
.
.
.
[Pt$_n$_1,Eta$_n$_1,Phi$_n$_1, M$_n$_1, obj$_n$_1]…

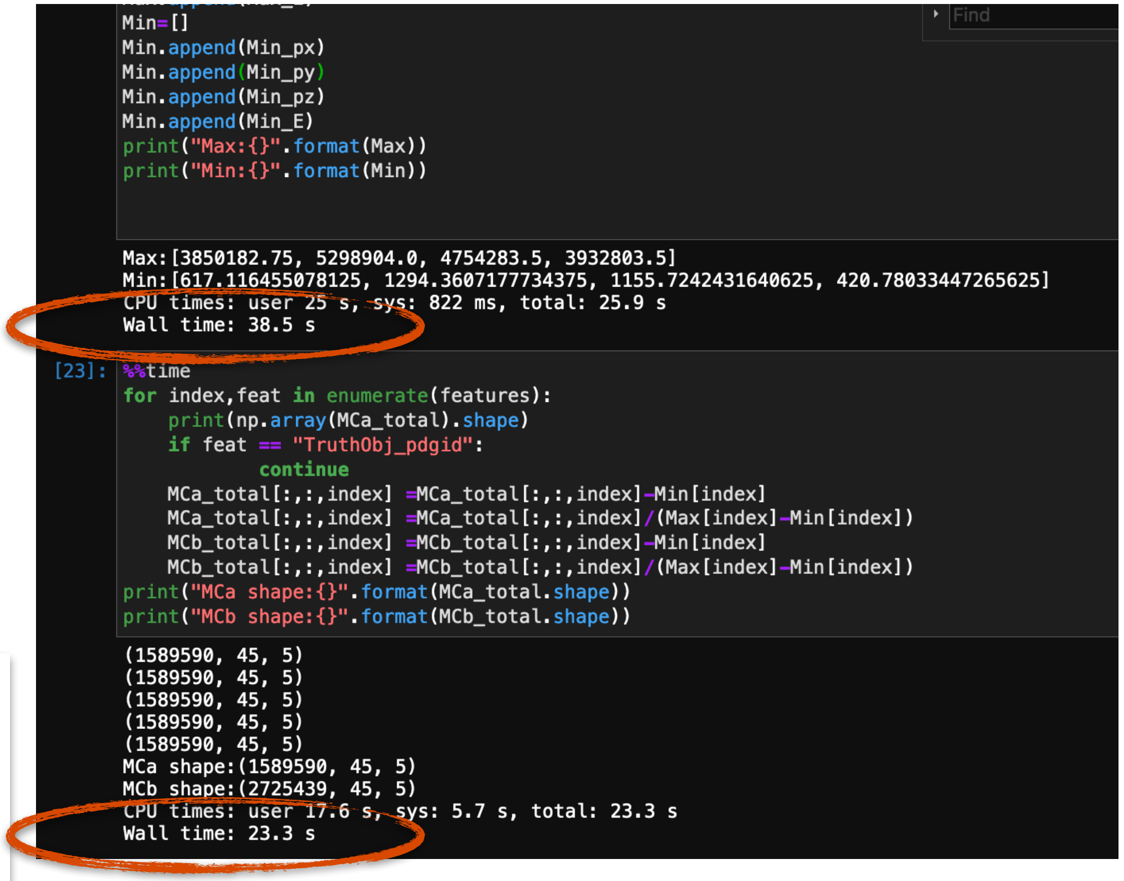
Computing performance in data preprocessing



→ 150 Workers

#(events, object, features):
(1589590, 45, 5)
(2725439, 45, 5)
CPU times: user 2min 39s, sys: 11.8 s, total: 2min 51s
Wall time: 3min 34s

preprocess_data



```
Min=[]
Min.append(Min_px)
Min.append(Min_py)
Min.append(Min_pz)
Min.append(Min_E)
print("Max:{}".format(Max))
print("Min:{}".format(Min))
```

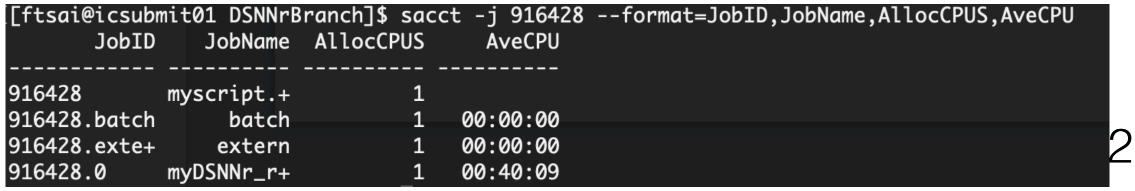
```
Max:[3850182.75, 5298904.0, 4754283.5, 3932803.5]
Min:[617.116455078125, 1294.3607177734375, 1155.7242431640625, 420.78033447265625]
CPU times: user 25 s, sys: 822 ms, total: 25.9 s
Wall time: 38.5 s
```

```
[23]: %%time
for index,feat in enumerate(features):
    print(np.array(MCa_total).shape)
    if feat == "TruthObj_pdgid":
        continue
    MCa_total[:,:,index] =MCa_total[:,:,index]-Min[index]
    MCa_total[:,:,index] =MCa_total[:,:,index]/(Max[index]-Min[index])
    MCb_total[:,:,index] =MCb_total[:,:,index]-Min[index]
    MCb_total[:,:,index] =MCb_total[:,:,index]/(Max[index]-Min[index])
print("MCa shape:{}".format(MCa_total.shape))
print("MCb shape:{}".format(MCb_total.shape))
```

```
(1589590, 45, 5)
(1589590, 45, 5)
(1589590, 45, 5)
(1589590, 45, 5)
(1589590, 45, 5)
MCa shape:(1589590, 45, 5)
MCb shape:(2725439, 45, 5)
CPU times: user 17.6 s, sys: 5.7 s, total: 23.3 s
Wall time: 23.3 s
```

→ Features scaling took only about 39+23 seconds.

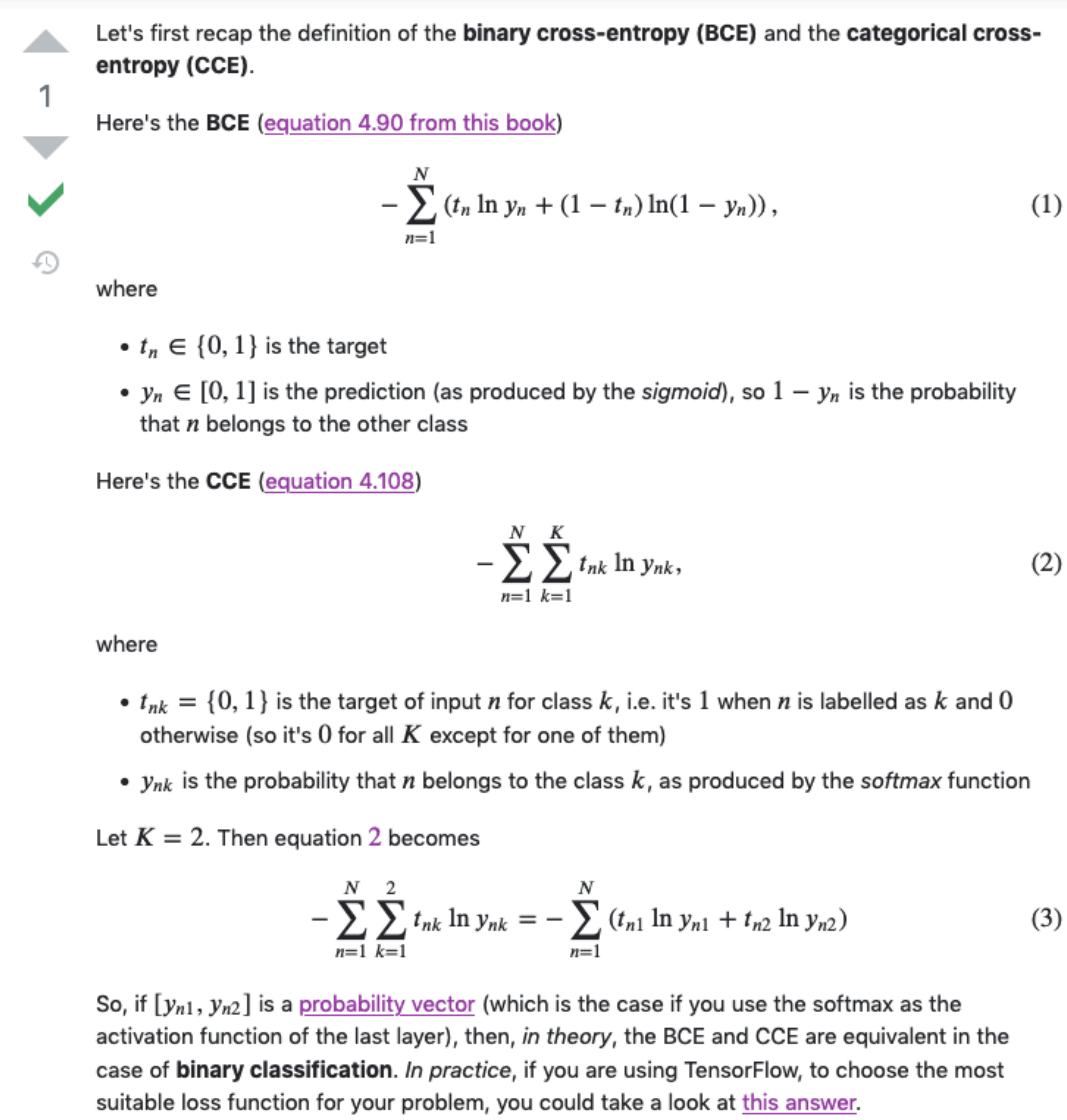
→ The parallel data processing can be done in 5-6 mins (the final amount of memory occupied ~ 47.4GB), while the standard data processing takes at least 40 mins (+memory usage is limited).



```
[ftsai@icsubmit01 DSNNrBranch]$ sacct -j 916428 --format=JobID,JobName,AllocCPUS,AveCPU
       JobID    JobName  AllocCPUS     AveCPU
------------ ---------- ---------- ----------
916428       myscript.+          1
916428.batch      batch          1   00:00:00
916428.exte+     extern          1   00:00:00
916428.0     myDSNNr_r+          1   00:40:09
```

BCE vs CCE

- source: the BCE and CCE are equivalent in the 2 classes case.



Let's first recap the definition of the **binary cross-entropy (BCE)** and the **categorical cross-entropy (CCE)**.

Here's the **BCE** (equation 4.90 from this book)

$$-\sum_{n=1}^{N}\left(t_n \ln y_n + (1-t_n)\ln(1-y_n)\right), \tag{1}$$

where

- $t_n \in \{0,1\}$ is the target
- $y_n \in [0,1]$ is the prediction (as produced by the *sigmoid*), so $1-y_n$ is the probability that n belongs to the other class

Here's the **CCE** (equation 4.108)

$$-\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk}, \tag{2}$$

where

- $t_{nk} = \{0,1\}$ is the target of input n for class k, i.e. it's 1 when n is labelled as k and 0 otherwise (so it's 0 for all K except for one of them)
- y_{nk} is the probability that n belongs to the class k, as produced by the *softmax* function

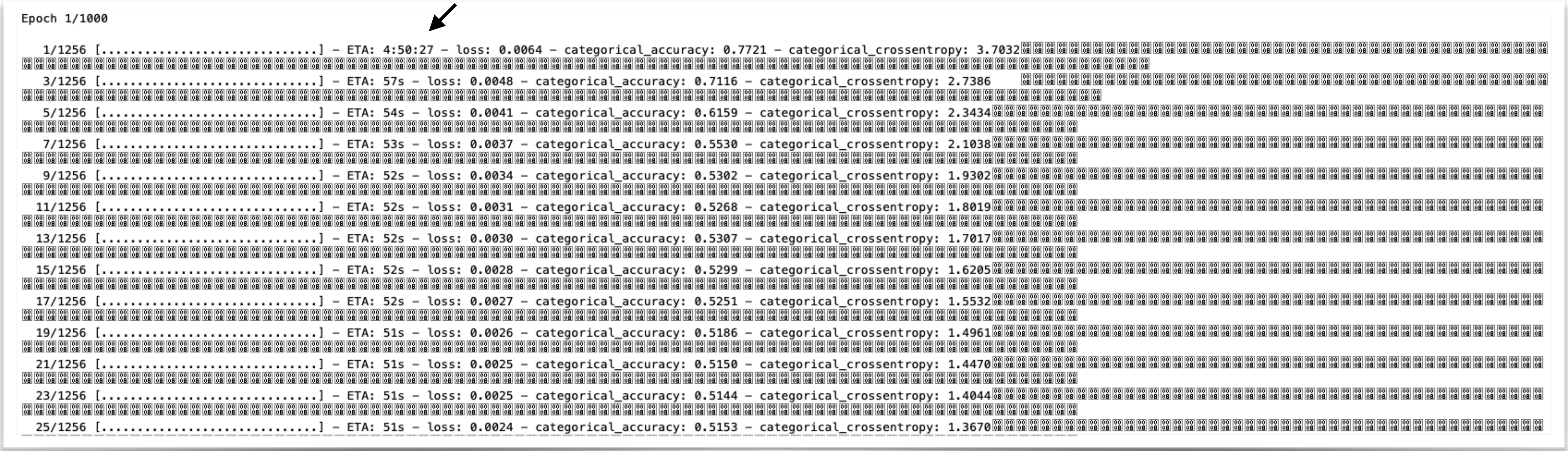
Let $K=2$. Then equation 2 becomes

$$-\sum_{n=1}^{N}\sum_{k=1}^{2} t_{nk} \ln y_{nk} = -\sum_{n=1}^{N}\left(t_{n1} \ln y_{n1} + t_{n2} \ln y_{n2}\right) \tag{3}$$

So, if $[y_{n1}, y_{n2}]$ is a probability vector (which is the case if you use the softmax as the activation function of the last layer), then, *in theory*, the BCE and CCE are equivalent in the case of **binary classification**. *In practice*, if you are using TensorFlow, to choose the most suitable loss function for your problem, you could take a look at this answer.

Training from epoch to epoch

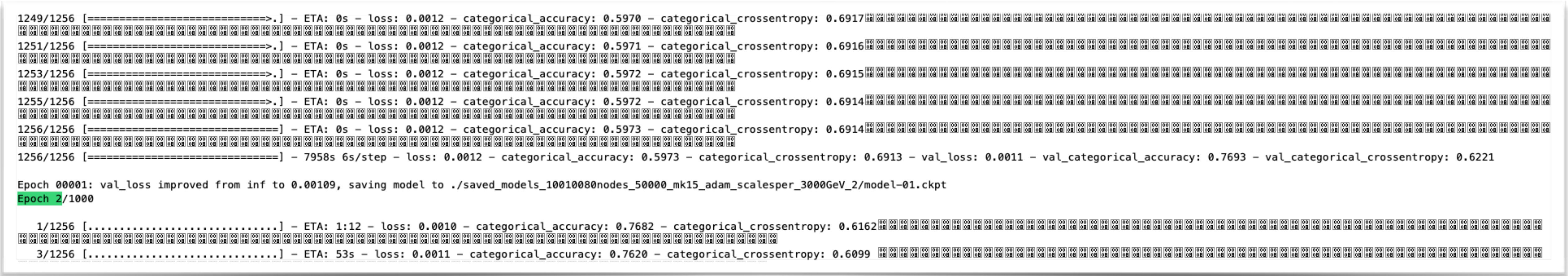
It took so long because the system's error is high due to random weights at the beginning.



```
Epoch 1/1000

   1/1256 [..............................] - ETA: 4:50:27 - loss: 0.0064 - categorical_accuracy: 0.7721 - categorical_crossentropy: 3.7032
   3/1256 [..............................] - ETA: 57s - loss: 0.0048 - categorical_accuracy: 0.7116 - categorical_crossentropy: 2.7386
   5/1256 [..............................] - ETA: 54s - loss: 0.0041 - categorical_accuracy: 0.6159 - categorical_crossentropy: 2.3434
   7/1256 [..............................] - ETA: 53s - loss: 0.0037 - categorical_accuracy: 0.5530 - categorical_crossentropy: 2.1038
   9/1256 [..............................] - ETA: 52s - loss: 0.0034 - categorical_accuracy: 0.5302 - categorical_crossentropy: 1.9302
  11/1256 [..............................] - ETA: 52s - loss: 0.0031 - categorical_accuracy: 0.5268 - categorical_crossentropy: 1.8019
  13/1256 [..............................] - ETA: 52s - loss: 0.0030 - categorical_accuracy: 0.5307 - categorical_crossentropy: 1.7017
  15/1256 [..............................] - ETA: 52s - loss: 0.0028 - categorical_accuracy: 0.5299 - categorical_crossentropy: 1.6205
  17/1256 [..............................] - ETA: 52s - loss: 0.0027 - categorical_accuracy: 0.5251 - categorical_crossentropy: 1.5532
  19/1256 [..............................] - ETA: 51s - loss: 0.0026 - categorical_accuracy: 0.5186 - categorical_crossentropy: 1.4961
  21/1256 [..............................] - ETA: 51s - loss: 0.0025 - categorical_accuracy: 0.5150 - categorical_crossentropy: 1.4470
  23/1256 [..............................] - ETA: 51s - loss: 0.0025 - categorical_accuracy: 0.5144 - categorical_crossentropy: 1.4044
  25/1256 [..............................] - ETA: 51s - loss: 0.0024 - categorical_accuracy: 0.5153 - categorical_crossentropy: 1.3670
```

⋮



```
1249/1256 [============================>.] - ETA: 0s - loss: 0.0012 - categorical_accuracy: 0.5970 - categorical_crossentropy: 0.6917
1251/1256 [============================>.] - ETA: 0s - loss: 0.0012 - categorical_accuracy: 0.5971 - categorical_crossentropy: 0.6916
1253/1256 [============================>.] - ETA: 0s - loss: 0.0012 - categorical_accuracy: 0.5972 - categorical_crossentropy: 0.6915
1255/1256 [============================>.] - ETA: 0s - loss: 0.0012 - categorical_accuracy: 0.5972 - categorical_crossentropy: 0.6914
1256/1256 [==============================] - ETA: 0s - loss: 0.0012 - categorical_accuracy: 0.5973 - categorical_crossentropy: 0.6914
1256/1256 [==============================] - 7958s 6s/step - loss: 0.0012 - categorical_accuracy: 0.5973 - categorical_crossentropy: 0.6913 - val_loss: 0.0011 - val_categorical_accuracy: 0.7693 - val_categorical_crossentropy: 0.6221

Epoch 00001: val_loss improved from inf to 0.00109, saving model to ./saved_models_10010080nodes_50000_mk15_adam_scalesper_3000GeV_2/model-01.ckpt
Epoch 2/1000

   1/1256 [..............................] - ETA: 1:12 - loss: 0.0010 - categorical_accuracy: 0.7682 - categorical_crossentropy: 0.6162
   3/1256 [..............................] - ETA: 53s - loss: 0.0011 - categorical_accuracy: 0.7620 - categorical_crossentropy: 0.6099
```